Synthesis of Optimal 1-Hot Coded On-Chip Controllers For BIST Hardware[1]

Debaditya Mukherjee, Charles A. Njinda,
Melvin A. Breuer

CENG Technical Report: 91-20

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4472

November 8, 1991

[1]This work was supported by the Defense Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under Contract No. JFB190092.

Abstract: *We present a procedure for merging on-chip controllers for BIST circuitry to reduce hardware overhead. We pick the 1-hot code state assignment and implicitly search the space of minimum prime compatible state covers to obtain an optimal merged controller. This procedure uses knowledge of the greatest lower bounds on states, arcs, next-state and the output logic of the merged controller to prune the search space.*

Contents

1 Introduction

Designers use DFT and BIST testable design methodologies (TDMs), such as LSSD and BILBO [1, 2] to obtain testable chips. The structural aspects of the TDM can be modelled by a *template.* Such a template conveys information about the type of structure (the *kernel*) to which the TDM is applicable, the source of the test vectors, the destination of the responses and also the path taken by these test vectors from source to destination. A *test schema* formalizes the execution characteristics of a TDM. When a TDM is *embedded* in an actual circuit, the *test schema* is customized to the specific circuit and is then called a *test plan.* More details on these concepts can be found in [3].

A circuit can have TDMs of the same or different types. Each such embedding gives rise to a *test plan.* The nature of the TDM as well as the complexity of the data transfer paths (I-paths, S-paths, T-paths [4] and F-paths[5]) between source and destination influences the complexity of the test plans. A test plan is in essence a *control strategy* for an embedding. The representation of a test plan in terms of a directed graph translates directly into the state transition graph of a finite state machine (FSM) controller for that embedding.

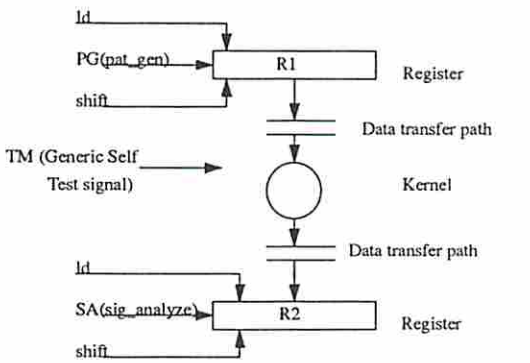
A testable VLSI chip can have numerous embeddings, which implies that there will be at least as many test plans and some of these plans can be quite complex. Embeddings in ASICs are usually not self contained structures. They share test sources and/or destinations and/or data transfer paths. These test plans will all have to be controlled keeping in mind that test plans can share resources. Thus the control of all these test plans can lead to a complex design problem. To date there is no methodology that, given a specific VLSI chip and a set of DFT/BIST TDMs, synthesizes a control entity (which may have sub-entities) with the objective of minimizing area overhead, functional delay, and at the same time meet some constraints on test time.

Previous work on the synthesis of test controllers focuses on specific aspects of the synthesis problem[6, 7, 8, 9, 10]. Beenker *et al.*[6] deal with the synthesis of a hierarchical test controller. Their approach uses separate and distributed custom controllers. To keep the synthesis task simple, standard interfaces are used for communication between the control units. This leads to designs with high area overhead. The optimality of test control line distribution has been investigated by Beausang and Albicki[7] but the optimality of the overall test controller has not been discussed. A procedure is given for the synthesis of on-chip controllers, but the controller area is not minimal[8] . Van Riessen *et al.*[9] present the design and implementation of a hierarchical testable architecture using boundary scan. This approach uses distributed controllers and the controller area is not optimal. The problem of merging test controllers to reduce

controller area overhead has been discussed by Marinissen[10]. He uses dummy or No-Op states and heuristic techniques for minimizing the number of states in the merged controller. However the impact of a minimized realization on the next state and output logic is not addressed.

1.1 Focus of this report

Figure 1 gives the structural template for the BILBO TDM. The template is a modified version of that given in [3]. It is assumed that all BIST registers have shift capability. Therefore both registers R1 and R2 have a signal called *shift*. *PG* is a signal that enables the pattern generation mode in R1 and *SA* enables the signature analysis mode in R2. Since, in most cases, these registers are also functional registers, they have a signal *ld* that is used for either loading or holding the registers in normal mode of operation. The existence of a generic test mode signal *TM* is assumed. The truth table defines the functional relationship between the various signals. For example, when *TM* is 0, the *ld* signal has control over the registers. When *TM* is 1, a precedence relationship is present between the *shift* and the PG/SA signals. When TM is 0 the circuit is in *normal mode*, and it is in *test mode* when TM is 1. Figure 2 shows a test plan for



Truth table for test register

TM	ld	PG/SA	shift	Register Function
0	1	x	x	load
0	0	x	x	hold
1	x	x	1	shift
1	x	1	0	pat gen/sig analyze
1	x	0	0	hold

Figure 1: The BILBO TDM structure

a particular embedding where there is a pattern generator, a signature analyzer and no other registers between them. The data transfer path may typically have multiplexers that need to be controlled in test mode. The test plan is a directed graph G={Head,Tail,V,E} where V is a set

of vertices, E is a set of edges and *Head* and *Tail* are also vertices (see Section 2.1). Both the edges and the vertices have labels. The labels for the edges specify the conditions for a transition between a pair of vertices. Each of the vertices in V have two sets of labels A_1 and A_2. Members of A_1 are all the control lines that have to be set to 1, and members of A_2 have to be set to 0. Unspecified control lines are don't cares. *Test length* is the number of test vectors needed to test a kernel. *Test plan length* is the number of vertices in V, referred to as phases, in a test plan.

In Figure 2, V={1,2}. E= { (Head,1),(1,2),(2,Tail),(2,1)}. For vertex 1, A_1={PG} and A_2={SA}, assuming that the data control path has no other logic on it. For vertex 2, A_1={SA} and A_2={PG}. Other control signals can also be activated if necessary.

Figure 3 shows an example circuit. The circuit has three combinational blocks (kernels) $L1$,$L2$ and $L3$ to be tested. $PG1$, $PG2$ and $PG3$ are pattern generators and SA is a signature analyzer. The *ld* signal on each unmodified (no pattern generation/signature analysis) register and the *select* lines on the multiplexers are activated by the functional controller in normal mode and by the test controller(s) in test mode. Test plans 1,2 and 3, corresponding to the three embeddings of the BILBO TDM, are shown in Figure 4. In *Test Plan 1*, the transition from Head to phase 1 occurs when all registers have been initialized. At this time SC is set to 1 for one clock period. In *phase 1* of the test plan PG1 is activated by c_1 and SA is in the hold

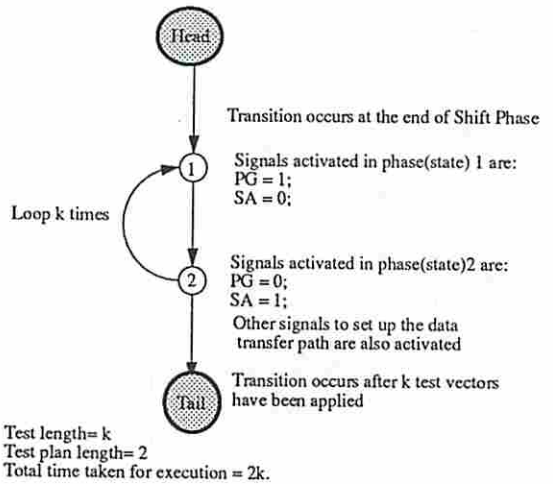


Figure 2: Test plan for a BILBO TDM embedding

mode($c_{10} = 0$). In *phase 2* the test vector x_{inp} generated by PG1 is loaded into R1. In *phase 3*, the response vector x_{out} from $L1$ is loaded in R1 through MUX1. In *phase 4*, x_{out} is loaded in

R2 from R1 through MUX3. In *phase 5*, SA captures x_{out} from R2 through MUX4. The test plan sequences through the 5 phases k times, where k is the *test length*. A signal TC is set to 1 when all tests have been applied. Test plans 2 and 3 operate in a similar way.

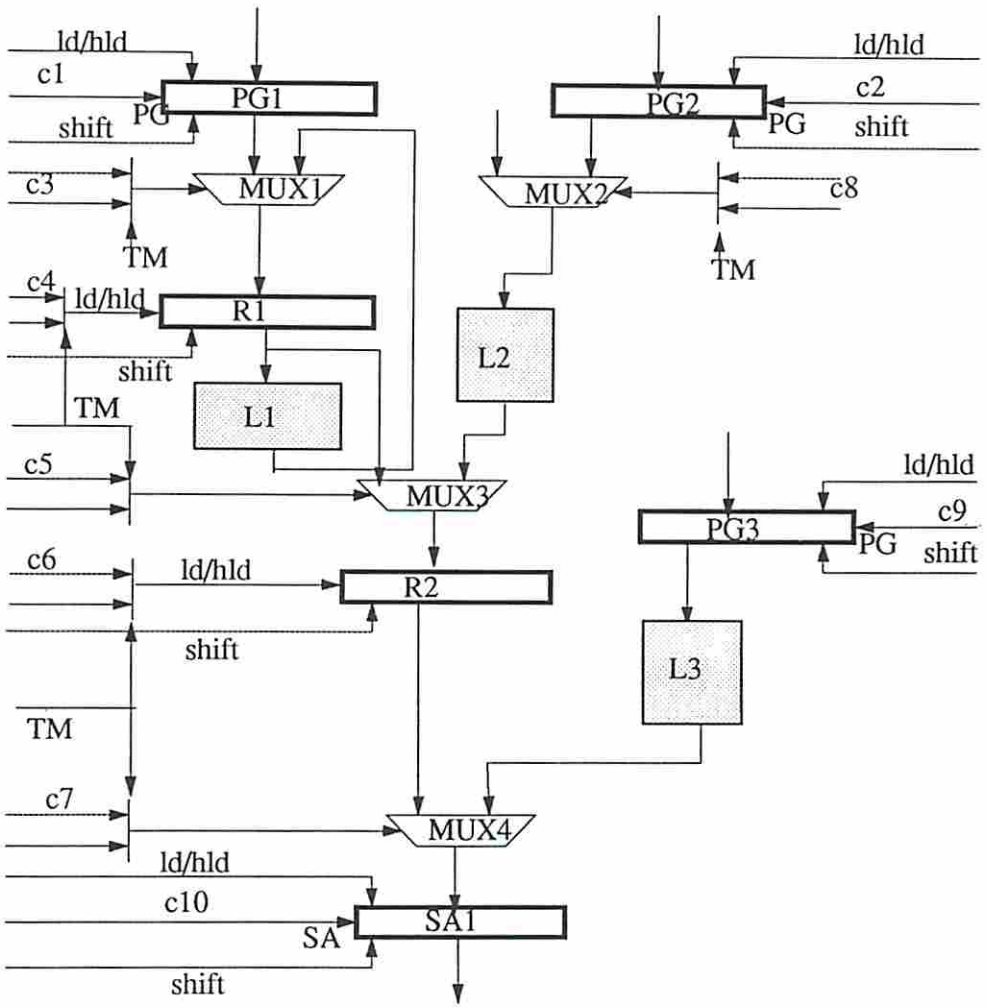


Figure 3: Example 1

For each kernel in a design there will be a test controller that executes a test plan like the ones illustrated. However, the test plans do not perform the set up task for the test environment e.g., initialize the BILBO registers or scan out the final signature. The test controller for a kernel

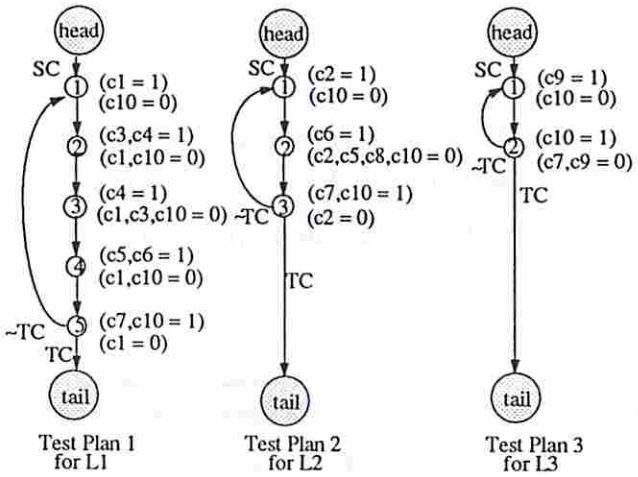


Figure 4: Test plans 1,2, and 3 for example 1

will thus not only have to execute the actual test plan shown, but will also have to set up the test environment. Since set-up is a common feature of all test plans, we can have a master controller that performs all the housekeeping functions and hands over control to simpler controllers to execute different test plans.

In this paper we attempt to solve the problem of merging these simpler controllers or submachines in order to minimize the complexity of the control logic. We will focus on the BILBO TDM and assume that test plans are executed serially, i.e. one after the other. Each test plan is assumed to be non-pipelined. The methodology is applicable to other TDMs (such as the SCAN TDM) and to test plans that operate in a pipelined manner.

2 The Test Environment

We next define the environment in which our test controllers will operate.

- There are n test plans to be executed.
- The *Test Plan length* or simply the *length* of test plan i will be denoted by *l(i)*.
- There are three counters on the chip.
  - There is a *test counter* (TCNTR) that is on a scan path and a value is scanned in this counter prior to the execution of each test plan. This value specifies the test length

of plan i. TCNTR has an input DEC-TC that decrements the counter. There is a signal TC that is true when the test counter reaches a value of 0.

- There is a *shift counter* (SCNTR) that is not on the scan path. It is reset at the start of the execution of a test plan(RST-SC). It has an input INC-SC that increments the counter and generates a output SC when the counter reaches a value equal to the length of the shift chain.
- A *test plan counter* (TPCNTR) that specifies the test plan being executed. This counter is reset by a signal (RST-TPC) and incremented by a signal (INC-TPC). When TPCNTR reaches the value n, it generates a signal TPC which indicates that all test plans have been executed. This counter is not on the scan chain. The output of this counter is fully decoded.

- A Test-Mode(TM) signal that is 0 in *normal mode* and is 1 in *test mode.*
- Test data are stored off chip.

Figure 5 shows the test controller, the counters and the functional controller for a testable circuit.

2.1 The Test Controller Model

The state diagram of the overall test controller that sets up the test environment and executes each of the n test plans is shown in Figure 6. It is modelled as a Moore machine, M, represented by the 5-tuple $(I, O, S, \delta, \lambda)$. I is the set of input lines, O the set of output lines, S the set of states, δ the state transition function and λ is the output function.

- I=$\{$TC,SC,TPC,TM,$tp_1, tp_2, ..., tp_n\}$
  where tp_i is activated for test plan i.
- O=$\{c_1, c_2, c_3, ..., c_k$,INC-SC,DEC-TC,INC-TPC,RST-SC,RST-TPC,SHIFT$\}$
  where $c_1, c_2, c_3, ..., c_k$ are the set of control lines controlling the circuit under test.
- S=$\{Idle_1, Idle_2$, Head, Tail, $S_a, S_b, S_{1_1}, ..., S_{1_{l(1)}}, S_{2_1}, ..., S_{2_{l(2)}}, ..., S_{n_1}, ..., S_{n_{l(n)}}\}$

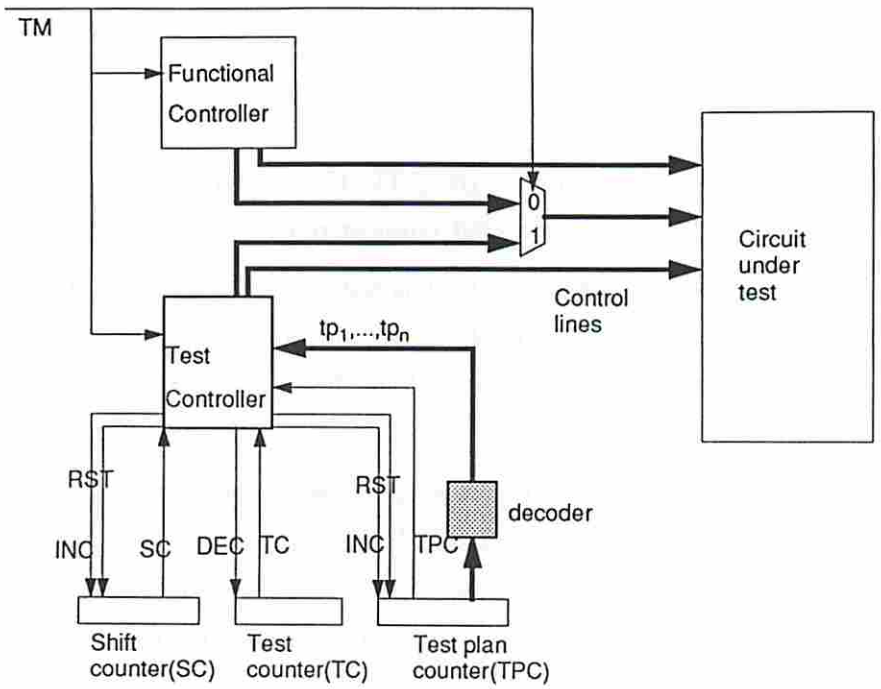


Figure 5: The control model

- δ: I_{val} x S $\rightarrow$ S | I_{val} set of all input values in the space spanned by I
- λ : S $\rightarrow$ O_{val} | O_{val} set of all output values in the space spanned by O

In normal mode, when TM=0, M stays in state $Idle_1$. When TM=1, M goes to S_a enabling RST-TPC and RST-SC to reset TPCNTR and SCNTR. Then M remains in state Head until shift in(out) is completed. On completion of initial shift (SC=1) M goes to the state corresponding to *phase 1* of test plan 1. M moves through the phases of test plan 1 and in the last phase makes a decision either to loop back to phase 1 or go to the Tail. This decision is based on the value of TC. If M is in Tail, and if all the test plans have not been executed (TPC=0), then M goes back to Head and shifts in the new seed for test plan 2 and shifts out the result for test plan 1. If TPC=1 and M is in Tail, then it moves to S_b and shifts out the results of the last test plan. Then M moves to $Idle_2$ and finally goes to $Idle_1$ when TM=0.

This machine can be decomposed into two parts. One part consisting of states $Idle_1$,$Idle_2$, Head, Tail, S_a and S_b, deals with the set up process and is common to all test plans. The second part deals with the actual execution of the test plans. We can therefore model the overall controller as interacting submachines M_{common} and M_{tp_i},i=1,...,n, where M_{tp_i} executes test plan i. Our primary focus will be on merging the submachines M_{tp_i} into one FSM.

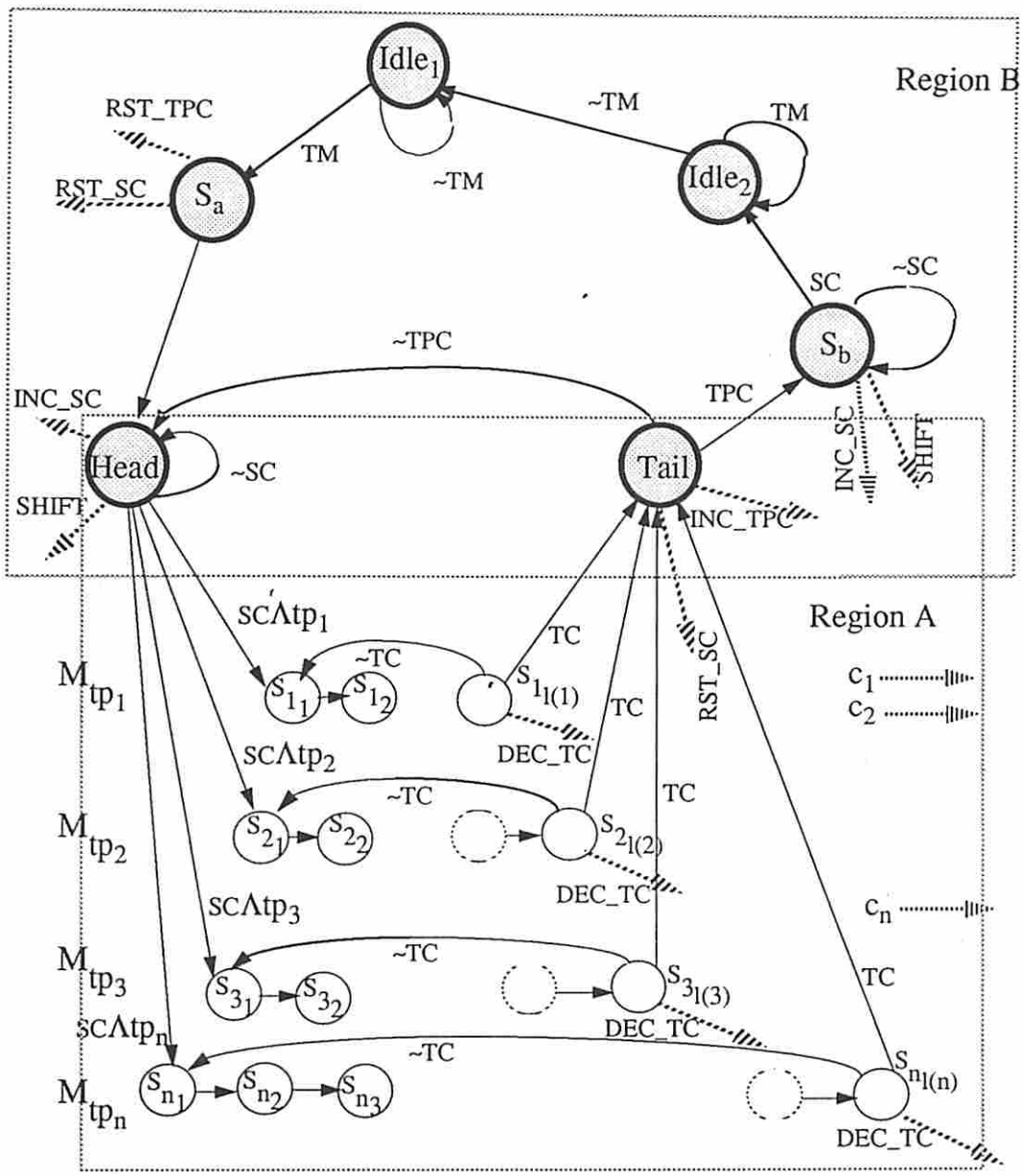


Figure 6: The overall test controller state diagram

2.2 Implementation details of the submachines

M_{common} can be implemented using any classical technique. In addition to the Head and Tail states, M_{tp_i} has q_i states. These states will be implemented using the 1-hot encoded state assignment, (one of the many choices in state assignment) and thus q_i flip-flops are required. From the examples used in this research, we have observed that the output logic for a 1-hot coded controller is trivial. Each control line is either driven directly by a flip-flop or at most by an OR(NOR) gate. We have thus restricted the output logic such that each output is driven at most by one OR/NOR gate. This restriction helps to prune the search space and from the examples we have looked at, this restriction is quite realistic. However, in general, using existing logic minimizers such as Espresso[11] or MIS-II[12] to compute the output logic may give better results when the total output logic is more than 1 two input OR(NOR) gate. But the procedure that is presented here is flexible enough to accomodate either a 1-level synthesizer (which we have used) or a 2-level synthesizer or a multilevel synthesizer. Our output optimality is defined in terms of 1-level output synthesis. We could as well define this output logic optimality in terms of 2-level synthesis or multilevel synthesis and use appropriate synthesizers.

2.2.1 Advantages of a 1-hot test controller

Some advantages of a 1-hot code over an assignment using a minimum number of flip-flops are listed below.

- The flip-flops in the controller can be distributed throughout the circuitry to reduce control wire routing overhead. In a multiphase test plan, test data and results move from one register to another in each phase. Each of these registers need to be controlled. There also might be data selectors such as multiplexers and sometimes complex combinational logic such as ALUs between two sets of registers that need to be controlled. A 1-hot coded controller for a test plan with q phases has q flip-flops. If the I-path used by a multiphase test plan has no cycles, then the number of functional registers in the I-path is equal to the number of phases in the test plan. The existence of one or more cycles in an I-path implies that at least one register is used more than once in the test plan. Each of these functional registers can be increased in length by one or more bits and these extra bits can be used in the 1-hot controller. The test control lines are thus generated very close to the modules that are being controlled rather than being generated by a test controller containing a minimum number of flip-flops, and routed all over the design. In a datapath dominated design, increasing the length of a register by one bit is more likely to be less

expensive than implementing a controller with standard cell flip-flops and routing control wires across the chip.

- All cycles in the controller can be broken by appropriately controlling the primary inputs. This simplifies the sequential test generation problem.
- Output logic, if any, is fully tested during the execution of the test plans. The controller generates a walking 1 pattern in test mode. This constitutes a complete test set for output logic synthesized with OR/NOR gates.
- If the test controller is made part of the scan path, then the 1-hot coded controller provides an efficient means of executing scan tests to supplement BILBO test.

It has been shown in [13] that there exists a close relationship between the synthesis approach chosen for an FSM and its testability. Appropriate state assignments enhanced the testability of some benchmark FSMs and resulted in a hardware overhead of 0%-30%. overhead. In [14] a new test algorithm for FSMs synthesized using the 1-hot code state assignment has been presented. The paper presents area, delay and test coverage data for a set of MCNC benchmark FSMs that are synthesized using the 1-hot code assignment and also synthesized using minimum number of flipflops. MUSTANG [15] is used in the latter case. All flip-flops in the minimum flip-flop FSMs have been made scannable. The FSMs have been placed and routed. It is interesting to note that, with the exception of one example, all the 1-hot coded FSMs have less area and delay than the minimum flip-flop ones.

To reflect the 1-hot coded state assignment, we can further refine the definition of the transition functions of the submachines M_{tp_i} as follows:

- δ_{entry} : (SC,Head) $\rightarrow S_{i_1}$ | S_{i_1} is called the *start* state of M_{tp_i}.
  Each (a,b) represents an ordered pair where a is the input and b is the present state.
- δ_{exit} : (TC,$S_{i_{l(i)}}$) $\rightarrow$ Tail | $S_{i_{l(i)}}$ is called the *end* state of M_{tp_i}.
- δ_i : $S_{i_j} \rightarrow S_{i_{j+1}}$ for j=1, ..., $l(i) - 1$
- $\delta_{feedback}$:($\overline{TC}$,$S_{i_{l(i)}}$) $\rightarrow S_{i_1}$

3 Merging 1-hot encoded machines

Recall that the test plans are to be run sequentially. The sequencing and activation of the submachines depends on the value of the test plan counter. Intuitively it appears that, instead of implementing each of the submachines as individual entities, we may save logic by sharing logic among the submachines. This sharing, which can be done via a merging process, will primarily result in the sharing of memory elements. The degree of sharing will depend on the degree of interaction between the control lines activated by the submachines.

We will now formulate *the merging problem:*
Given n submachines controlling n test plans, obtain one machine M_{mrg} that has, (1) Head and Tail as its entry and exit states, (2) controls each of the test plans in turn in accordance to the input from the test plan counter, and (3) has a 1-hot coded implementation with minimum next state logic and output logic. The portion of the state diagram in Region A of Figure 6 satisfies

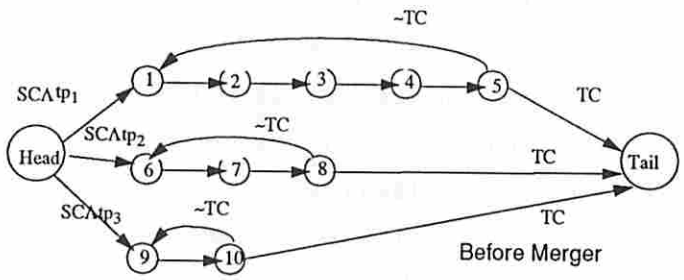


Figure 7: An example of an unoptimized merger

conditions (1) and (2). In our running example, Figure 7 represents the state transition graph of M_{mrg}. The number of states in this merged machine is $\sum_{i=1}^{n} q_i$, where $n = 3$. Each of these states activate some control lines. If all the states of a test plan are compatible with all the states of other test plans, then Figure 8 is an example of a M_{mrg} which satisfies conditions (1) and (2) and has minimum next state logic.

Since satisfying the first two conditions in the merging problem is trivial, we will focus our attention on satisfying the third condition. The M_{mrg} of Fig 6 will be used as a starting point. The state transition table of this machine is shown in Figure 9. The unspecified outputs appear as x in the table.

We will use the pair chart technique to find all pairs of compatible states. Some observations are helpful.

- The states of any one M_{tp_i} are mutually incompatible.
- It is not necessary to make multiple passes over the pair chart as is required for general machines. This is because for a pair of output compatible states i and j, an input tp_x specifies the next state only for i or j but not both.

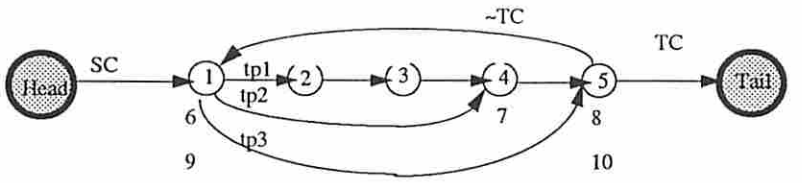


Figure 8: An example of a best case merger

	TP1	TP2	TP3	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
1	2			1	x	x	x	x	x	x	x	x	0
2	3			0	x	1	1	x	x	x	x	x	0
3	4			0	x	0	1	x	x	x	x	x	0
4	5			0	x	x	x	1	1	x	x	x	0
5	1			0	x	x	x	x	x	1	x	x	1
6		7		x	1	x	x	x	x	x	x	x	0
7		8		x	0	x	x	0	1	x	0	x	0
8		6		x	0	x	x	x	x	1	x	x	1
9			10	x	x	x	x	x	x	x	x	1	0
10			9	x	x	x	x	x	x	0	x	0	1

Figure 9: The transition table for example 1

Figure 10 is the pair chart where only the entries in the polygon are of interest. The other entries denote compatibility between states in the same test plan and as mentioned above, states in the same plan are mutually incompatible. In the pair chart Y stands for a compatibility and X denotes an incompatibilty.

From the pair chart the *maximum compatibles* (MCs) and the *prime compatibles* (PCs) can be found[16]. The PCs for this problem are the same as the MCs because the *class sets* of the MCs are null sets. Figure 11 shows the 2-compatibles, 3-compatibles and the PCs.

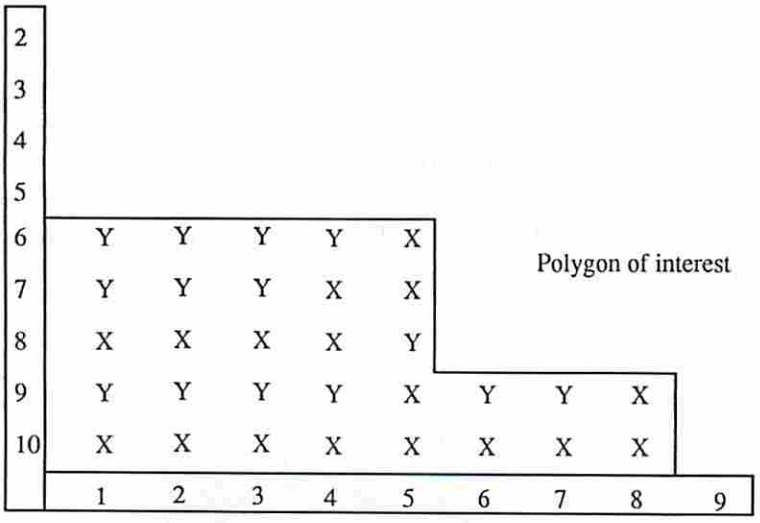


Figure 10: The pair chart

In a classical state minimization technique the next step would be to find *one* minimal PC cover for the set of states in the state transition table[16]. Then state, input and output encodings are determined so as to minimize the resulting logic implementation[11, 17, 15, 18]. Herein lies the major difference between classical techniques and ours. *We have already decided to use the 1-hot code.* To find the simplest hardware realization, it is necessary to consider *all* the minimal PC covers for this machine. We shall show later that the actual search space in most cases is, in fact, larger than the space of all minimal PC covers.

2 compatibles :

1-6, 1-7, 1-9 2-6, 2-7, 2-9 3-6, 3-7, 3-9 4-6, 4-9 5-8 6-9, 7-9

3-compatibles :

1-6-9, 1-7-9, 2-6,-9, 2-7-9, 3-6-9, 3-7-9, 4-6-9 ,

PC's :: 1-6-9, 1-7-9, 2-6-9, 2-7-9, 3-6-9, 3-7-9, 4-6-9, 5-8, 10

Figure 11: The compatibles and prime compatibles

To illustrate how the choice of a minimal covers affects the implementation of our merged 1-hot coded controller, consider the six possible combinations of PCs shown in Figure 12, all of which give a minimal cover. Figures 13(a) and (b) are the state transition graphs of the merged controller corresponding to two different choices of minimal covers. Hardware implementation of the two graphs shows that the implementation of the transition graph of Figure 13(a) has less

logic than the one for Figure 13(b). The output logic is the same in both cases. Counting the edges(arcs) of the transition graphs shows that Figure 13(a) has less arcs then Figure 13(b). We will show in the next section that there is a direct relationship between the number of arcs in the transition graph of M_{mrg} and cost of next state logic. In Section 4 we present a branch and bound procedure that builds a transition graph of the minimal merged machine.

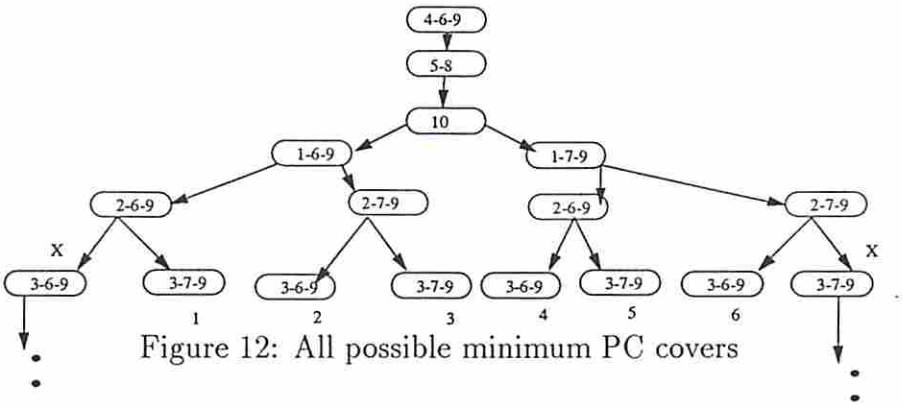


Figure 12: All possible minimum PC covers

3.1 Greatest lower bound of the number of states, arcs, next state logic and output logic

Recall that M_{tp_i} has q_i states, for i=1,...,n. Two cases need to be considered.

1. Each of the q_i are distinct and can be ordered in a strictly decreasing manner, i.e. $q_1 > q_2 > q_3 ... > q_n$.

2. The q_i are not distinct and the states can be ordered in a non-increasing manner, e.g. $q_1 = q_2 = ... = q_{t_1} > q_{t_1+1} = q_{t_1+2} = ... = q_{t_2} > ... > q_{t_{s-1}+1} = q_{t_{s-1}+2} = ... = q_{t_s}$ The M_{tp_i} with equal number of states are grouped together. In case 2, there are s such (state) groups. For case 1, s=n.

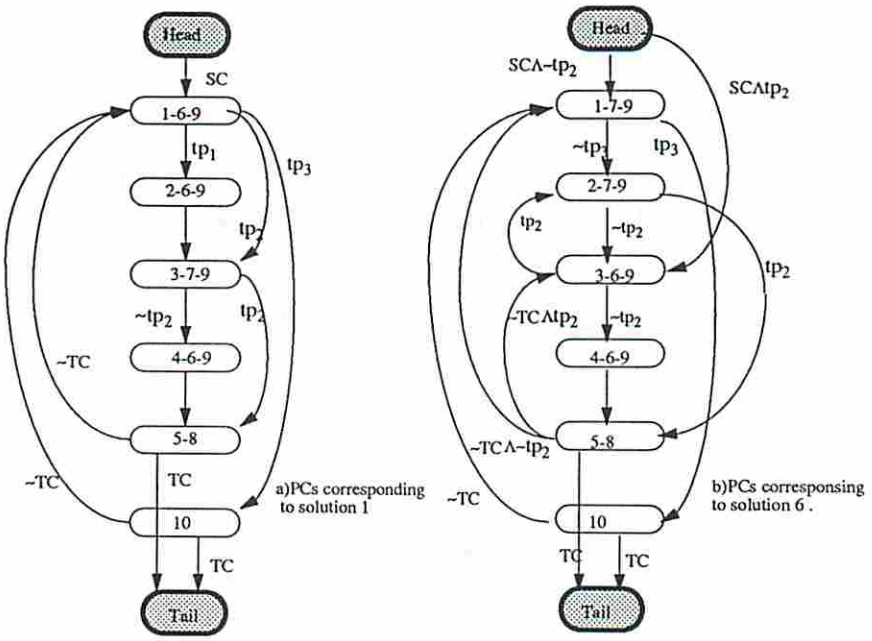


Figure 13: State transition graphs for two choices of minimum covers

3.1.1 Greatest lower bound on number of states in M_{mrg}

Proposition 1 For either of the above two cases, the greatest lower bound *inf* of the number of states in M_{mrg} is $max(q_i)$ which, in terms of the ordered sequences above, is q_1.

We next state a theorem that gives the *greatest lower bound* of the number of arcs in (the state transition graph of) M_{mrg}. We will then establish a relationship between the number of arcs and the implementation complexity of the next state logic of M_{mrg}.

3.1.2 Greatest lower bound of arcs and next state logic of M_{mrg}

Theorem 1 The *inf* of the number of arcs in the state transition graph of M_{mrg} is q_1+s+1, where q_1 is $max(q_i)$ and s is the number of state groups.

Proof: The proof is by induction. Without loss of generality we can assume that the submachines have been ordered in a non-increasing sequence with respect to the number of states and subscripted as 1,2,...,n.

Let n=1. Then s=1. There are $q_1 - 1$ arcs connecting S_{1_1} to $S_{1_{q_1}}$. There is one arc from the *Head* to S_{1_1}, one from $S_{1_{q_1}}$ to *Tail* and the last one is the feedback arc connecting $S_{1_{q_1}}$ to

S_{1_1}. Since n=1, M_{tp_1} and M_{mrg} are the same. Thus the total number of arcs is q_1+2 = q_1+s+1.

For n=2, s could be either 1 or 2. s=1 implies that both M_{tp_1} and M_{tp_2} have the same number of states. Assuming that there are no incompatibility between any pair of states in M_{tp_1} and M_{tp_2}, both submachines can be represented by the same state transition graph, in which case the total number of arcs is q_1+s+1. The situation s=2 is interesting because the two submachines to be merged have unequal number of states. By our ordering principle, $q_1 > q_2$. Now q_1+2 = q_1+s arcs are *required* for M_{tp_1}. When we merge M_{tp_2} to M_{tp_1} we will have to add *at least* one additional arc over and above the arcs already used for M_{tp_1}. This arc is necessary to distinguish between the longer and the shorter test plans. Thus the lower bound on the number of arcs is q_1+s+1.

Assume that the bound holds for some $n = k-1$. We will show that the same bound holds when we add another submachine. There are two cases. If q_k equals q_{k-1} then no additional arcs need be added, and since the lower bound for $n = k-1$ was q_1+s+1, and s has not changed, the lower bound for $n = k$ is correct. If $q_k > q_{k-1}$, then we need to add *at least* one arc to the merged machine built up so far. Thus the lower bound is q_1+s+1 where the arc to be added is reflected in the value of s.

This proves that $q_1 + s + 1$ is a lower bound. Given a set of machines M_{tp_i}, i=1,...,n, we can construct a merged machine M_{mrg} having $q_1 + s + 1$ arcs. Thus $q_1 + s + 1$ is the *inf.* □

Corollary 1 Merging starting states (S_{i_1}) to form S_{mrg_1} and merging ending states $S_{i_{q_i}}$ to form $S_{mrg_{q_1}}$ for i=1,...,n is a *necessary* condition for achieving the *inf* of arcs in M_{mrg}.

Proof: Only the outline of the proof for n=2 and $q_1 > q_2$ will be given. If the start and end states are not merged together, then three cases need to be considered. 1) Start states are not merged but end states are merged. 2) End states are not merged but start states are merged. 3) Both start and end states are not merged. Recall the proof of *Theorem 1* and note that case 1 requires at least *two* additional (entry and feedback for tp_2) arcs, case 2 requires at least *two* additional arcs (feedback and exit for tp_2) and case 3 requires at least *three* additional arcs (entry, exit and feedback for tp_2). □

We can think of transition arcs between two states being broken up into two parts, the outgoing arc (from a state) and the incoming arc (to a state). The existence of a single outgoing transition arc from a state implies that no logic is needed for the implementation of that outgoing arc. However when a state has more than one outgoing transition arc, then each transition

is a function of some subset of $\{tp_1, ..., tp_n\}$, and the *greatest lower bound* of logic needed to "implement" each outgoing arc is a product term of 2 literals. We will assign a cost of 1 to a product of 2 literals. So each outgoing arc has a cost of at least 1. A product of k literals has cost $k-1$. To implement two incoming transition arcs to a state we need the sum of 2 literals.. Assign a cost of $k-1$ to a sum of k literals. Thus the cost of implementing k incoming arcs to a state is $k-1$.

Proposition 2 Let M_{mrg} and M'_{mrg} be any two merged machines corresponding to different choices of minimal covers. If the number of arcs of M'_{mrg} is greater than the number of arcs of M_{mrg} , then the cost of implementing the next state logic of M'_{mrg} is greater than the cost of implementing the next state logic of M_{mrg}.

Proof: Assume that the above statement is false. Then the cost of implementing the next state logic of M'_{mrg} is less than or equal to the cost of the next state logic of M_{mrg}. Without loss of generality assume that the number of arcs of M_{mrg} equals the *inf* of arcs. Assume that it is also possible to add an additional arc to M_{mrg} to obtain another machine M'_{mrg} that corresponds to a valid merger. The number of arcs in M'_{mrg} is greater than the number of arcs in M_{mrg} and the cost of the next state logic of M'_{mrg} should be less than or equal to the cost of the next state logic of M_{mrg}.

Since only one additional arc has been added to M_{mrg}, this arc *must* be an outgoing arc from some state S_{mrg_i} and an incoming arc for some state S_{mrg_j}, where $i < j$. Consider the following cases: (1) the state S_{mrg_i} had only one outgoing arc before the new arc was introduced; and (2) there were y outgoing arcs before the introduction of the new arc. For case (1), the introduction of the new arc will result in the addition of 2 units of logic cost for implementing the outgoing arcs and an additional 1 unit of logic cost for the incoming arc for state S_{mrg_j}. For case (2) consider the two sub cases, (a) $s = n$ and (b) $s < n$. Consider subcase (a). Assume that x outgoing arcs from various states preceeding S_{mrg_i} have already accounted for (differentiated between) x submachines. Therefore the cost of implementing the y outgoing arcs at S_{mrg_i}, before addition of the extra arc was, $min((n-x), 2(y-1))$. After addition of the new arc, the cost is $min((n-x+1), 2y)$. Clearly, the cost of the outgoing arcs at S_{mrg_i} increases after the addition of the new arc. There is also a 1 unit increase in the cost of implementing the incoming arc for state S_{mrg_j}. Using the concepts presented in the proof of Theorem 2 we can also show that there is a cost increase for subcase (b). Thus for all the cases, the cost of implementing the next state

logic of M'_{mrg} is greater than that for M_{mrg}. This contradicts our original assumption. Therefore Proposition 2 is true. □

Theorem 2 The following two conditions are *sufficient* for achieving the minimum cost next state logic in M_{mrg}: (1) the number of arcs in M_{mrg} equals the *inf* of arcs, and (2) all the outgoing transition arcs that distinguish various test groups, s, emerge from the same state S_{mrj_i} for some i=1,...,q_1-2.

Proof: We have $tp_1, ..., tp_n$ available. Assume that complements are also available. Let $k_1, k_2, ..., k_s$ be the number of test plans in the s state groups. Let $k_1 \geq k_2 \geq ... \geq k_s$. Let r be the number of groups with only one member, $0 \leq r \leq s$. Consider first the case when all the outgoing transition arcs emerge from the same state. Let L_{same} be the total cost of implementing the outgoing arcs. Then

$$L_{same} = min(n, \sum_{i=2}^{s} k_i + 2(s-1) - r) \qquad (1)$$

Let the s groups come from some t states in M_{mrg}. Thus the s groups are divided into t sets, $V_1, V_2, ..., V_t$. Let p_i be the number of groups assigned to V_i and r_i be the number of groups in V_i having only one test plan each. Thus $0 \leq r_i \leq p_i$, $i = 1, ..., t$. $V_i \cap V_j = \oslash, i \neq j$.

Note that the set V_t will contain exactly one group i with k_i, i=1,...,s, test plans. Thus the state M_{mrg_j} which corresponds the set V_t, has only one outgoing arc and this arc does not need any logic for its implementation. Also if M_{mrg_i} corresponds to the set V_{t-1}, then M_{mrg_i} "comes before" M_{mrg_j} and $i < j$.

The total logic needed to implement all these outgoing arcs is
$L_{diff} = min\{n, \sum_{k_i \in V_1} k_i + 2p_1 - r_1\} + min\{(n - \sum_{k_i \in V_1} k_i), \sum_{k_i \in V_2} k_i + 2p_2 - r_2\} +$
$min\{(n - \sum_{k_i \in V_1 \cup V_2} k_i), \sum_{k_i \in V_3} k_i + 2p_3 - r_3\} + ... + min\{(n - \sum_{k_i \in V_1 \cup ... \cup V_{t-2}} k_i, \sum_{k_i \in V_{t-1}} k_i + 2p_{t-1} - r_{t-1}\}$

If $(n \leq \sum_{k_i \in V_1} k_i + 2p_1 - r_1)$ then $L_{diff} > n > L_{same}$ because L_{same} is at most n,
else if $((n > \sum_{k_i \in V_1} k_i + 2p_1 - r_1) \wedge ((n - \sum_{k_i \in V_1} k_i) \leq \sum_{k_i \in V_2} k_i + 2p_2 - r_2))$ then $L_{diff} > n > L_{same}$,
else if $((n > \sum_{k_i \in V_1} k_i + 2p_1 - r_1) \wedge ((n - \sum_{k_i \in V_1} k_i) > \sum_{k_i \in V_2} k_i + 2p_2 - r_2) \wedge ((n - \sum_{k_i \in V_1 \cup V_2} k_i) \leq \sum_{k_i \in V_3} k_i + 2p_3 - r_3))$ then $L_{diff} > n > L_{same}$,

.

.

.

else if $((n > \sum_{k_i \in V_1} k_i + 2p_1 - r_1) \wedge ((n - \sum_{k_i \in V_1} k_i) > \sum_{k_i \in V_2} k_i + 2p_2 - r_2) \wedge ... \wedge ((n - \sum_{k_i \in V_1 \cup ... \cup V_{t-2}} k_i) \leq \sum_{k_i \in V_{t-1}} k_i + 2p_{t-1} - r_{t-1}))$ then $L_{diff} > n > L_{same}$
else $L_{diff} = \sum_{k_i \in V_1 \cup ... \cup V_{t-1}} k_i + 2\sum_{i=1}^{t-1} p_i - \sum_{i=1}^{t-1} r_i$.

Lets look at the last case in more detail and try to minimize L_{diff}. $\text{Max}\sum_{i=1}^{t-1} r_i = s - 1$ and $min(\sum_{k_i \in V_1 \cup ... \cup V_{t-1}} k_i)$ occurs when k_1 is allocated to V_t. Now $\sum_{i=1}^{t-1} p_i = s - 1$. Therefore L_{diff} in this case is $\sum_{i=2}^{s} k_i + (s-1)$.

The minimum value of the term $\sum_{i=2}^{s} k_i + 2(s-1) - r$ in L_{same} occurs when $r = s-1$. Thus $L_{same} = min(n, \sum_{i=2}^{s} k_i + (s-1))$. Comparing L_{same} and L_{diff} we conclude that $L_{diff} \geq L_{same}$.

The cost of implementing the incoming arcs is the same in both cases. Thus the complexity of next state logic when all the arcs that distinguish different groups of test plans emerge from the same state is less than or at most equal to the complexity of arcs emerging from different states. Also from *Proposition 2* we know that a M_{mrg} with number of arcs equal to the *inf* of arcs has lower next state logic cost than a M_{mrg} with number of arcs greater than the *inf* of arcs. Thus we have proved *Theorem 2.* □

Corollary 2 Let $k_1, k_2, ..., k_s$ be the number of test plans in the s state groups, where these groups are ordered such that $k_1 \geq k_2 \geq ... \geq k_s$. Let r be the number of groups which have only one test plan each. Then the minimum cost of the next state logic, denoted by C_{nsl}, is given by the following formula.

$$C_{nsl} = min\{n, \sum_{i=2}^{s} k_i + 2(s-1) - r\} + s + 3$$

Proof: From equation (1) in the proof of *Theorem 2*, the cost of implementing the outgoing arcs excluding arcs ((Head,S_{mrg_1}),($S_{mrg_{q_1}}$,S_{mrg_1}),($S_{mrg_{q_1}}$,Tail)) is $min(n, \sum_{i=2}^{s} k_i + 2(s-1) - r)$. The cost of implementing the incoming arcs for the s groups is *s-1*. The cost of implementing the three arcs mentioned above is 4. □

3.1.3 Greatest lower bound of output logic of M_{mrg}

The state transition table of the machines M_{tp_i}, i=1,...,n, shows the values of the output lines c_j corresponding to the present state for every machine. Consider the part of the transition table

corresponding to only one machine M_{tp_i}. Let c_j, j=1,...,p, be the set of output lines corresponding to this machine. c_j corresponds to a column vector with q_i entries. If there is one or less 1 entries (the rest may be 0's or x's) or one or less 0 entries (the rest may be 1's or x's) in the column vector corresponding to c_j, then c_j comes directly from a flip-flop or a fixed logic level and needs no output logic. However if there are more than one 1 entries *and* more than one 0 entries in any column, then the corresponding control line has to be driven by either an OR gate or a NOR gate. Let a be the number of 1s and b be the number of 0s in c_j, where $a, b > 1$. If $a > b$ then a b input NOR gate is needed, and if $b > a$, then an a input OR gate is needed. If the cost of a 2 input OR/NOR gate is 1, then the cost of gates in the two cases above is $b - 1$ or $a - 1$. When more than one c_j has $a, b > 1$, then sharing of gates between them may be necessary to get a minimal implementation.

As stated earlier, we have restricted our output logic to one level of OR/NOR gate implementation. This restriction is realistic because all the examples that we looked at have trivial output logic (for the 1-hot code implementation) and do not warrant using general 2-level or multiple level logic minimization algorithms.

Let OL be a q x p matrix where the columns correspond to the p control lines and rows to the q states. All columns where $a \leq 1$ or $b \leq 1$ are deleted. OL is now reduced to a q x p' matrix. A pair chart technique is used to find compatible columns. Each of the compatibles has a cost. Our problem is to find a set of compatibles that covers all columns and has minimum cost. Various techniques exist to solve this minimum cost cover problem.

Proposition 3 Let L_{ol_i} be the minimum output logic for 1-hot coded submachines M_{tp_i} i=1,...,n. Then the greatest lower bound on the output logic for M_{mrg} is $L_{ol_{mrg}}$=max $\{L_{ol_i} \mid$ i=1,...,n$\}$.

Proof: Suppose a control line c_i, is driven by only one machine. Then another machine is merged with this machine and c_i is now driven by a merged machine. The process of merging states between two state machines has either no effect on the the original number of 1s and 0s in a column vector representation of c_i, or adds 1s and/or 0s. The cost of implementing the control line c_i thus can never be less than when there was no merger. This is also true when sharing of logic between outputs is considered. In a merged machine, all submachines must retain their functionalities. Thus the lower bound of output logic in the merged machine is $max\{L_{ol_i} \mid i = 1, ..., n\}$.

It is conceivable that the merger of machines does not affect the 1 and/or 0 count of the outputs of machine M_{tp_i} where $L_{ol_i} = \max\{L_{ol_j} \mid j = 1, ..., n\}$. L_{ol_i} is a valid output logic complexity of M_{mrg}. Thus it is the greatest lower bound. □

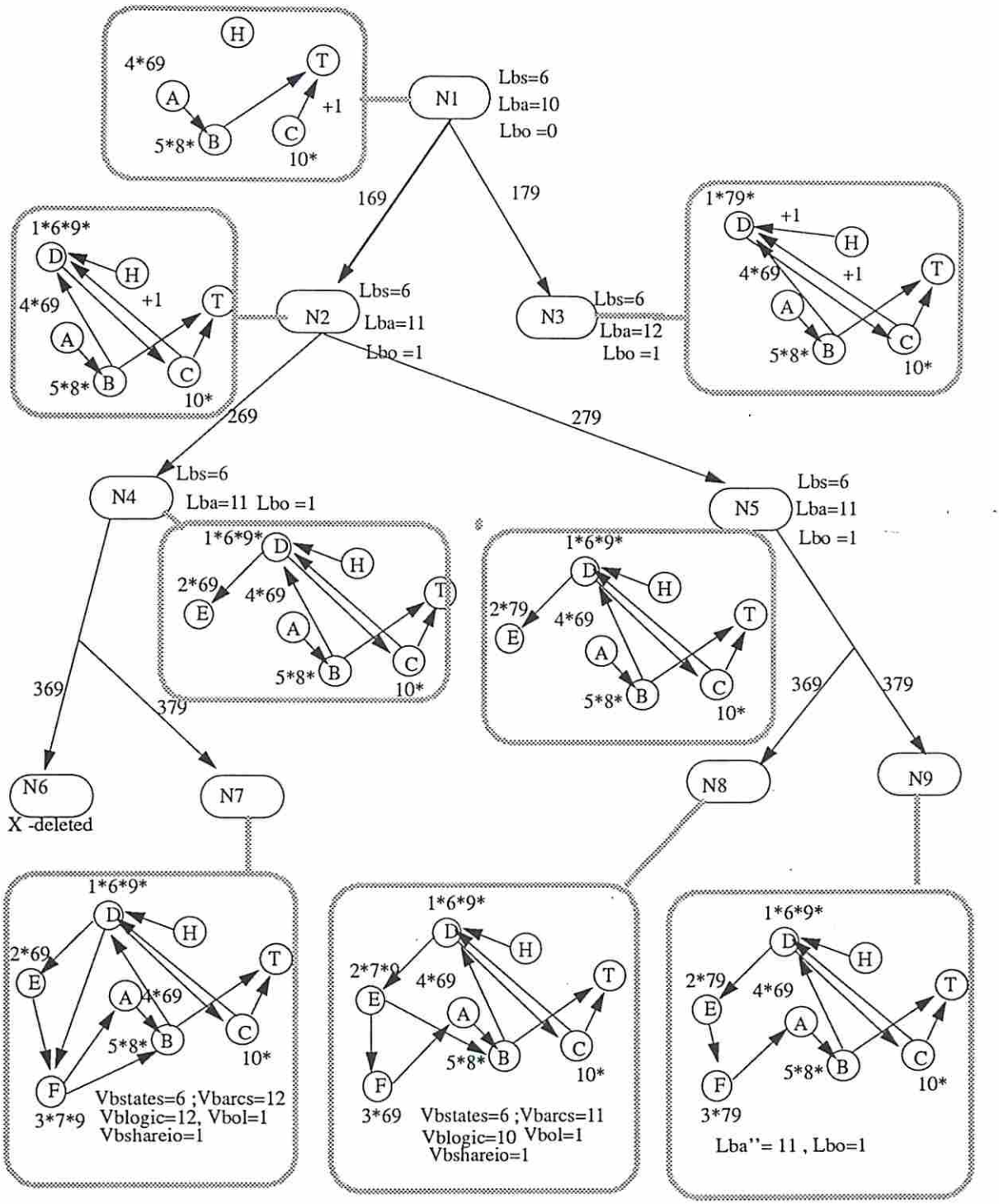
Sometimes it may be possible to share gates between input and output logic. This may happen when all the S_{q_i} are not merged in the same state in M_{mrg} and some control lines are activated by only those states in M_{mrg} where the S_{q_i} are assigned. There can be *at most s-1* such gate sharing possible. We take into account sharing of input/output logic only after we have looked at the next state logic and output logic separately.

4 Procedure SOHOT

We have developed an implicit enumeration procedure SOHOT (*S*ynthesis of *O*ptimal 1-*HOT* controllers) that incrementally builds up the transition graph of the merged machine, M_{mrg}, that has a minimum number of states and minimum cost input and output logic. The basic data structure of SOHOT is a n-ary tree called *Search_tree*. Each node N_i in *Search_tree* has some properties attached to it. One of these properties is a directed graph DG. DG=(V,E) represents the state transition graph (partial or complete) of a M_{mrg}.

We will present the basic concepts of SOHOT by illustrating how an optimal M_{mrg} is built for the example circuit given in Figure 3. The *Search_tree* along with the DGs for every node in this tree are illustrated in Figure 14. *Node_list* is a list of nodes in *Search_tree*. Initially *Node_list*=$\{N_1\}$. *Freq_list* is a list of states in M_{tp_i},i=1,2,3, sorted in nondecreasing order of their occurences in the PCs shown in Figure 11. States 4,5,8,10 have a frequency of 1. Thus the three PCs (4,6,9),(5,8) and (10) which cover these states *must* be used to build M_{mrg} and are selected first. These PCs are attached to the three new vertices, A,B and C created in the DG of N_1. Since the states 4,5,8,10 will not appear again, we *assign* them to the vertices A,B and C and mark this assignment by *starring* them. States 6 and 9 are not starred because they occur in other PCs. It is not known at this point if assigning 6 and 9 to vertex A will yeild an optimal M_{mrg}. However, states 4,5,6,8,9 and 10 are deleted from *Freq_list*. Since a transition exists between states 4 and 5 in M_{tp_1}, an edge is added between A and B. Similarly edges (B,T) and (C,T) are added to the DG of N_1 because transitions (5,Tail),(8,Tail) and (10,Tail) exist in all three M_{tp_i}.

The information from the DG of N_1 is used to compute metrics that establish a lower bound on the number of states (*lbs*), next state logic (*lba*) and output logic (*lbo*) of the M_{mrg}


H
4*69
A
T
+1
5*8*
B
C
10*
N1
Lbs=6
Lba=10
Lbo =0
169
179
1*6*9*
D
H
4*69
+1
T
A
5*8*
B
C
10*
N2
Lbs=6
Lba=11
Lbo =1
N3
Lbs=6
Lba=12
Lbo =1
1*79*
D
+1
H
4*69
+1
T
A
5*8*
B
C
10*
269
279
N4
Lbs=6
Lba=11 Lbo =1
N5
Lbs=6
Lba=11
Lbo =1
1*6*9*
D
H
2*69
E
4*69
A
T
5*8*
B
C
10*
2*79
369
379
N6
X -deleted
N7
N8
N9
2*7*9
3*7*9
3*69
3*79
F
Vbstates=6 ;Vbarcs=12
Vblogic=12, Vbol=1
Vbshareio=1
Vbstates=6 ;Vbarcs=11
Vblogic=10 Vbol=1
Vbshareio=1
Lba''= 11 , Lbo=1

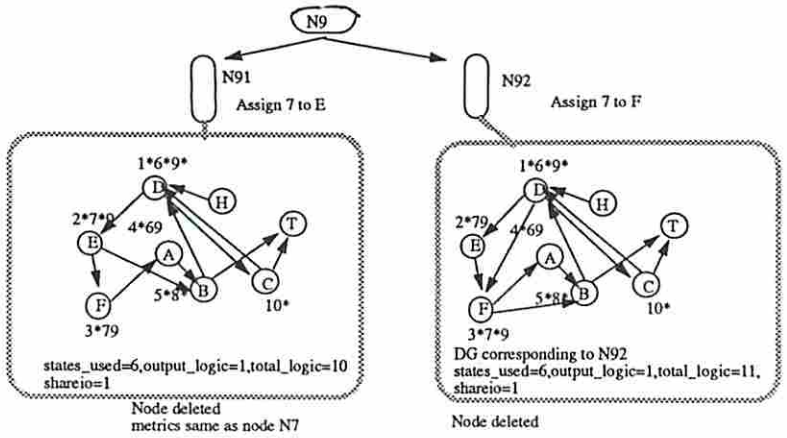


Figure 14: The search tree for example 1

whose partial transition graph is DG. Theoretically, the *inf* of arcs is 9 (*Theorem 1*). To achieve this result it is necessary to merge all the end states 5,8,10 (*Corollary 1*). However, in the DG of N_1, states (5,8) and 10 are assigned to two different vertices and there are two edges instead of one going to T. Thus the new lower bound, *lba*, is 10 (9+1). From *Proposition 1*, the *inf* of the number of states is 5. The DG of N_1 has 3 vertices if H and T are excluded. However, 3 states from M_{tp_1} and 1 state from M_{tp_2} have not been accounted for. Assignment of the 3 states of M_{tp_1} will require *at least* 3 additional vertices in DG. Hence *lbs* is 6. An output matrix OL is created having three rows corresponding to the three vertices A,B and C in DG. The columns correspond to the control lines activated by the states 4,5,8 and 10 assigned to these vertices (4 to A, 5 and 8 to B, 10 to C). No gate is required to implement the output logic. So *lbo* is 0. These metrics are properties of N_1. *Freq_list* and OL are also properties of N_1.

State 1, the first element of the modified *Freq_list* with a frequency of 2, needs to be assigned to a vertex. Since PCs (1,6,9) and (1,7,9) both cover state 1, N_1 is expanded to create children N_2 and N_3. These nodes inherit all the properties of N_1. Vertex D is added to the DGs of both nodes. PCs (1,6,9) and (1,7,9) are attached to the vertex D in the DGs of N_2 and N_3 respectively. State 1 is starred for both nodes. The edges of the DG of N_2 is updated next. Since 1 is a start state in M_{tp_1}, an edge (H,D) is added. Since merging the start states 1,6,9 is a necessary condition for achieving the *inf* of arcs in M_{mrg}, and since 6 and 9 appear in vertex D along with state 1, we assign 6 and 9 to D even though they appear in other PCs (and vertices). So 6 and 9 are starred. Edges (D,C),(B,D) and (C,D) are then added. The metrics *lba*,*lbs* and *lbo* need to be computed for this node. This DG has edges (B,D) and (C,D) corresponding to two feedback arcs instead of one required to achieve the *inf* of arcs. So *lba* is incremented by 1 and becomes 11. *lbs* of this node is 6, since 2 states in M_{tp_1} have yet to be accounted for. An additional row is added to the OL of N_2 to reflect vertex D and the entries in this row are the control lines activated by the states 1,6,9. Control line c_{10} requires a two input OR(NOR) gate,

so *lbo*=1. Similarly state 9 is starred and edges (H,D), (B,D),(C,D) and (D,C) are added to the DG of N_3. Similar to the DG of N_2, the DG of N_3 has two feedback arcs. In addition only two out of three start states are assigned to D, thus another arc will be needed to cover the (Head,6) transition in M_{tp_2}. Thus *lba* is incremented by 2 and becomes 12. *lbs* is 6 and *lbo is 1.*

The node N_1 is deleted from *Node_list* and N_2 and N_3 are added. N_2 is the next candidate for expansion since *lba* of N_2 is less than the *lba* of N_3. The *Freq_list* of N_2 only has states 2,3 and 7 since the other states have been deleted. State 2 has a frequency of 2, hence N_2 is expanded to create two children N_4 and N_5. PCs (2,6,9) and (2,7,9) are attached to vertex E in the DGs of nodes N_4 and N_5. State 2 is starred in both cases. Edges (D,E) are added to both. The metrics for both the nodes are 6,11 and 1 for *lbs*, *lba* and *lbo* respectively.

Node_list=$\{N_4, N_5, N_3\}$. Expansion of N_4 creates two nodes N_6 and N_7 corresponding to the two choices of PCs for state 3. Node N_6 is deleted because its *Freq_list* still contains state 7 whereas the *Freq_list* of N_7 is empty. Now all states are found to be starred in the DG of N_7 implying that all states have been assigned to vertices and this DG represents a complete transition graph of a M_{mrg}.

The next state logic, output logic, the number of states and the number of arcs used by the M_{mrg} corresponding to the DG of N_7 is determined. Sharing of logic (if any) between next state and output logic is taken into account and assigned to a local variable *shareio.* The shared logic, *shareio*, is subtracted from the sum of the logic needed to implement the next state and output logic and assigned to another local variable *Total_logic.* Global variables *Vbstates*, *Vbarcs*, *Vblogic*, *Vbol* and *Vbshareio* are assigned values for the states, arcs, total logic, output logic and shared logic respectively for the best DG found at any point in SOHOT. Since the DG of N_7 is the first complete transition graph found so far, *Vbstates*=6, *Vbarcs=12*, *Vblogic*=12, *Vbol*=1, *Vbshareio*=1 and *Best_Node*=N_7.

Node_list=$\{N_5, N_3\}$. The metrics of N_5 are compared against the best found so far and N_5 is expanded to produce N_8 and N_9 corresponding to the two PC choices for state 3. The DG of N_8 is updated and it is observed that it is also a complete transition graph of a M_{mrg}. This DG has less arcs and logic than the best found so far. So *Best_node*=N_8, *Vbstates*=6, *Vbarcs=11*, *Vblogic*=10, *Vbol*=1 and *Vbshareio*=1. After the DG of N_9 is updated, it is found that state 7 has not been starred even though it appears in vertices E and F. This implies that we have to evaluate the impact of the assignment of state 7 to each of these vertices. This makes the search space larger than the space of all minimal PC covers. *lba* is no longer a good metric for nodes such as N_9 which have unassigned states. Another metric *lba''* is calculated instead. This is a heuristic and is a lower bound on the number of arcs. It is more optimistic (gives smaller

numbers) than *lba* but helps to order nodes in *Node_list* if more than one node similar to N_9 exists. Assume that E and F are lumped together to form a new vertex EF. Then the transition (6,7) is covered by edge (D,EF) but no edge exists from EF to B to cover (7,8). The DG has 10 edges and *at least* 1 additional edge will be required. So *lba''* is 11. *lbs* is 6 and *lbo* is 1.

Node_list=$\{N_9, N_3\}$, N_9 is inserted before N_3 because *lbs* for both the nodes are equal and *lba''* of N_9 is less than the *lba* of N_3. Expansion of N_9 is different from the expansions encountered so far. Here two nodes N_{91} and N_{92} are created corresponding to the two vertices E and F to which state 7 can be assigned. Node N_{91} is deleted because *total_logic* equals *Vblogic* (we do not keep track of more than one solution with the best cost) and node N_{92} is deleted because *total_logic* is greater than *Vblogic*.

Node_list=$\{N_3\}$. Lets assume that the best implementation of a M_{mrg} built up from the DG of N_3 has 12 arcs and shares 1 unit of logic between input and output. However, the best M_{mrg} found so far has *Vbarcs*=11, *Vbol*=1 and *Vbshareio*=1. Thus expanding N_3 will never give a better result. So N_3 is deleted and the hardware realization of the best M_{mrg}, corresponding to N_8, is shown in Figure 15.

The procedure will terminate if at any leaf node it finds a DG whose next state and output logic equals the *inf*s of the next state and output logic respectively. This is a very powerful feature of SOHOT because it does not need to explore the solution space any further. The pseudocode for SOHOT is given in the appendix.

5 Results

Some results using SOHOT are presented in Figure 16. Ckt 1 is the example shown in Figure 3. The entry *Total # states* in the table corresponds to $\sum_{i=1}^{n} q_i$, where n is the number of test plans for a particular circuit. The *inf* of arcs, next state logic (*nsl*), output logic (*ol*) and states are also tabulated. It is important to note that the search space is not simply the space of all minimal PC covers. A state may occur in more than one PC in the minimal cover and an optimal assignment of this state to a PC increases the search space. For example, Ckt 1 has 6 PC covers, but the entire search space has 18 leaf nodes. As seen from the table, M_{mrg} for Ckt 1 has 11 arcs, 6 states, and 10 2-input gates, and the procedure only examined 4 leaf nodes out of a possible 18. Ckt 2 is interesting because the next state logic and output logic of M_{mrg} at one of the leaf nodes is equal to the *inf* of the next state logic and output logic, respectively, and the procedure terminated without searching any further. In this case, only 2 out of a possible 18 nodes were

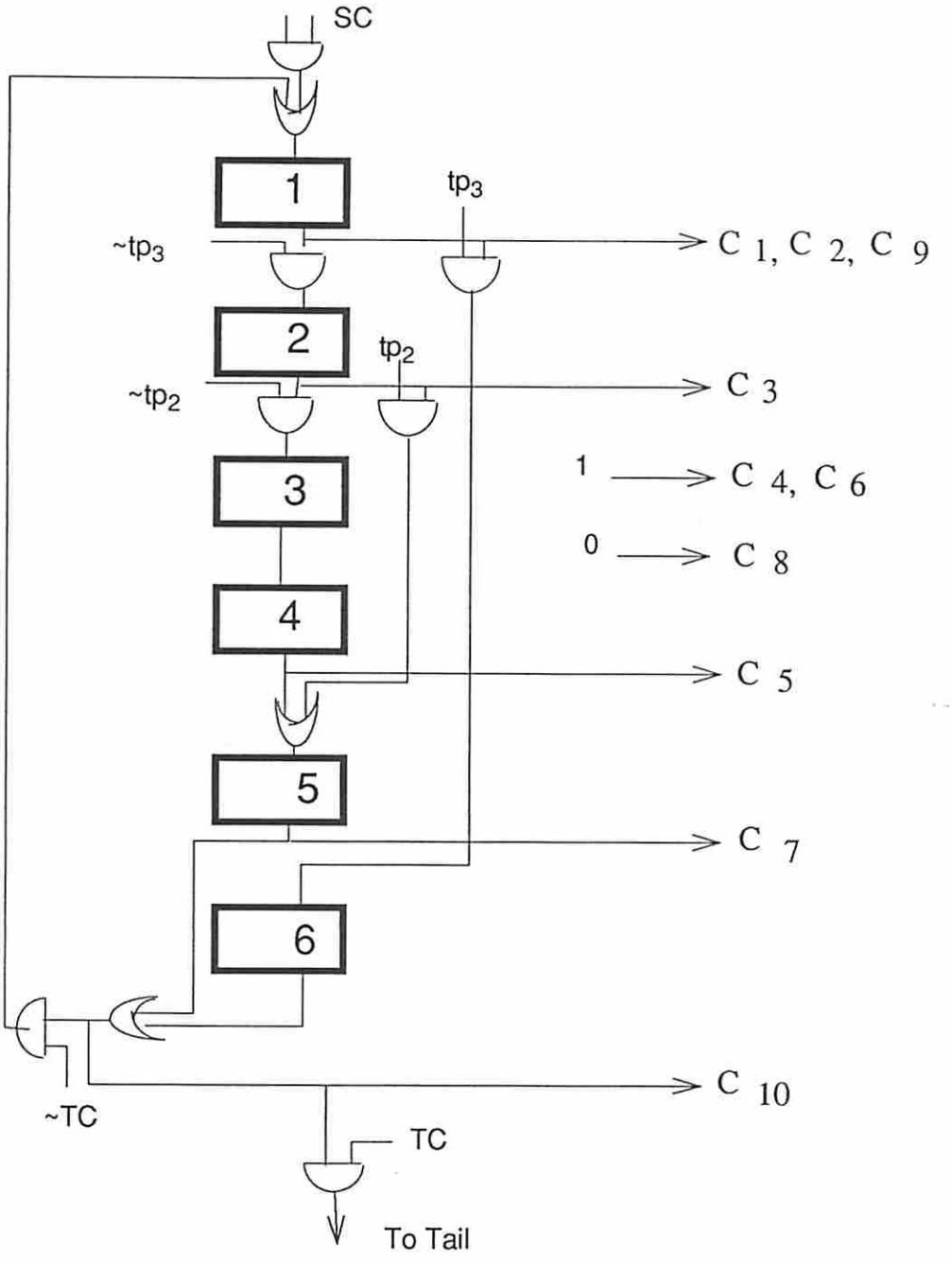


Figure 15: Hardware implementation of the minimal merged machine for example 1

examined. For Ckt 3, the procedure examined only 2 leaf nodes out of 10. Ckt 4 has only 1 answer. Except for an OR gate driving a control line in Ckt 1 and Ckt 3, all other control lines of the circuits are driven directly from the flip-flops of their merged machines.

							Optimal Merged Machine						
Ckt Name	Total # states	Inf Arcs	Inf nsl	Inf ol	Inf States	Leaf nodes in search space	# of Arcs	# of States	next state logic	output logic	shared logic	Total logic	Leaf nodes generated
Ckt 1	10	9	9	0	5	18	11	6	10	1	1	10	4
Ckt 2	15	10	9	0	6	18	10	6	9	0	0	9	2
Ckt 3	12	9	9	0	5	10	12	6	13	1	1	13	2
Ckt 4	13	8	7	0	5	1	9	5	10	0	0	10	1

Figure 16: Results

We have run our examples through a set of standard synthesis tools. For state minimization we used STAMINA, a tool developed at University of Colorado, Boulder. JEDI[18] is used for state assignment, and SIS, an updated vesion of MIS, has been used for logic minimization. STAMINA, however, produces a minimal state Mealy machine. Since we use the Moore model, in some cases the cardinality of the minimum state cover produced by STAMINA is smaller than that produced by SOHOT. However, this is the only state minimization tool that is integrated with SIS, which is a synthesis package developed at U.C. Berkeley. For each of the exmaple circuits we carried out state assignment using both the 1-hot code and the minimum number of flip-flops. Then logic minimization is performed using a standard script file. Figure 17 shows the results. The descriptions given to the synthesis tools did not have the *Head* and *Tail* states. Therefore when we obtained the literal counts for the designs implemented by SOHOT, we did not count the literals contributed by the transitions to/from the Head and Tail states. From Figure 17, we can see that the literal counts for designs obtained using SOHOT are much smaller

than those obtained by standard synthesis tools. The entries *lits sop* and *lits fac* represent the number of literals in the sum of product form and factored form respectively.

Circuit name	Total # states	SOHOT		STAMINA -> JEDI -> SIS						
					1-hot encoding			min-code encoding		
		min states(ffs)	lits	min states	# of ffs	lits sop	lits fac	# of ffs	lits sop	lits fac
Ckt 1	10	6	11	5	5	17	16	3	28	27
Ckt 2	15	6	8	6	6	16	14	3	31	30
Ckt3	12	6	14	5	5	18	16	3	23	22
Ckt4	13	5	9	5	5	21	21	3	24	24

Figure 17: Comparison with standard synthesis tools

6 Conclusion

Contemporary state machine synthesis techniques perform state minimization first and then do state, input and output encoding. However, in this paper we have shown that for some specific problems, such as the test controller synthesis problem, it is possible to use certain properties specific to the nature of the controllers and come up with an implicit enumeration technique to achieve a truly optimal synthesized machine. Some of the subproblems by themselves are NP-complete or NP-hard. For example, state minimization is a NP-hard problem. Determining the minimum output logic, even when it is restricted to one level, is an NP-complete problem. Since the problems in the test controller domain are of restricted size, in this paper we have assumed that we will solve these subproblems exactly. We could have an option of using heuristic techniques for obtaining the PCs or use existing logic minimizers to obtain the output logic. However, using the 1-hot code makes the output logic trivial and consequently the output logic problem size is small enough to make exact techniques feasible. Also note that the size of a PC in our problem is upper bounded by the number of machines, n, to be merged, and the complexity of finding all the PCs is $O((\sum_{i=1}^{n} q_i)^{2^n})$. The complexity is exponential in the number of machines being merged and not in the number of states. Therefore if we assume that the number of machines is much less than the total number of states, then the worst case complexity is actually

a polynomial function of the total number of states.

We have focused on the BILBO TDM and have assumed that the test plans are being executed serially. However the test controller model for the partial SCAN TDM is almost identical to the BILBO TDM, and SOHOT can be applied with minor modifications.

We mentioned in Section 2.2.1 that one of the main advantages of using a 1-hot coded controller is the flexibility of distributing the flip-flops in the controller throughout the design such that control wire routing overhead is reduced. The flip-flops of the merged controllers for the four circuits given in the result can indeed be distributed throughout the circuits. Also, the single OR gate in the output logic of Ckt 1 and Ckt 3 is fully tested by the walking 1 pattern that is generated in the controller during the test mode.

In this paper we have used the Moore model. Using a Mealy model may produce different results. We have also assumed that the inputs are available fully decoded and restricted ourselves to a single level of output gates. We plan to remove these restrictions in the future.

References

[1] M. S. Abadir and M. A. Breuer. A knowledge based system for designing testable VLSI chips. *IEEE Design and Test of Computers*, pages 56–68, August 1985.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, New York, NY, 1990.

[3] M. S. Abadir and M. A. Breuer. Test schedules for VLSI circuits having built-in test hardware. *IEEE Trans. on Computers*, pages 361–368, April 1986.

[4] M. A. Breuer, R. Gupta, and R. Gupta. AI aspects of TEST: A system for designing testable VLSI chips. In *IFIP Workshop on Knowledge-Based Systems for Test and Diagnosis*, pages 29–75, September 1988.

[5] S. Freeman. Test generation for data-path logic: The f-path method. *IEEE J. of Solid-State Circuits*, pages 421–427, April 1988.

[6] F. Beenker, R. Dekker, and R. Stans. A testability strategy for silicon compilers. In *Proc. Int'l Test Conf.*, pages 660–669, 1989.

[7] J. Beausang and A. Albicki. Towards determining an optimal test control line distribution scheme for a self-testable chip. Technical Report Tech. Rep. EL-86-04, Department of Electrical Engineering, The University of Rochester, March 1986.

[8] J. Beausang and A. Albicki. A methodology for designing self-testable VLSI chips: Synthesis. Technical Report Tech. Rep. EL-87-05, Department of Electrical Engineering, The University of Rochester, July 1988.

[9] R. P. van Riessen, H. G. Kerkhoff, and A. Kloppenberg. Designing and implementing an architecture with boundary scan. *IEEE Design and Test of Computers*, pages 9–19, February 1990.

[10] E. J. Marinissen. Automated test control block generation and minimization. Master's thesis, Eindhoven University of Technology, January 1990.

[11] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic, Boston, MA, 1984.

[12] R Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. Computer-Aided Design*, pages 1062–1081, November 1987.

[13] S. Devadas, H. K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Synthesis and optimization procedures for fully and easily testable sequential machines. In *Proceedinds, International Test Coference*, pages 621–630, September 1988.

[14] R. Z. Makki, S. Bou-Ghazale, and C. Tianshang. Automatic test pattern generation with branch testing. *IEEE Trans. on Computers*, pages 785–791, June 1991.

[15] 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. *IEEE Trans. Computer-Aided Design*, pages 1290–1300, December 1988.

[16] A. D. Friedman and P. R. Menon. *Theory and Design of Switching Circuits.* Computer Science Press, 1975.

[17] G. DeMicheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment of finite state machines. *IEEE Trans. Computer-Aided Design*, pages 269–285, July 1985.

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

A Appendix

A.1 Pseudocode for main procedure SOHOT

The following are global variables : (Freq_list, End_list, Start_list, Fback_list, PC, s)

Procedure **SOHOT**()

{

$Node\_list = \{N_1\}$;

$N_1.flag = 0$; $Root\_flag = 1$; $Best\_node = \emptyset$; $Vbarcs = \infty$; $Vbol = \infty$; $Vblogic = \infty$; $Vbstates = \infty$; $Vbshareio = 0$;

$Star\_list = \emptyset$;

$N_1.freq\_list =$ $Freq\_List$; $N_1.DG = (V \mid V = \{H, T\}, E \mid E = 0)$;

If($\exists\ j \in N_1.freq\_list \mid j.freq = 1$)

{$\forall\ j \in N_1.freq\_list \mid j.freq = 1$,

{ $N_1.DG.V = N_1.DG.V \cup create\_node(u)$;$u.states\_cov = PC_j$;

Star all $k \in u.states\_cov \mid k.freq = 1$; Star_list=all starred states; Delete $u.states\_cov$ from $N_1.freq\_list$;

}

$N_1.DG.E = Process\_Edges(N_1.DG)$;

If ($N_1.freq\_list == \emptyset$) return $N_1.DG$;

$N_1.OL = Update\_output\_logic(N_1.DG)$;$N_1.lba = Compute\_arcs(N_1.DG)$;

$N_1.lbo = Compute\_ouput\_logic(N_1.DG)$;

}

else {$N_1.lba = 0$; $N_1.lbo = 0$; }

While($Node\_list \neq \emptyset$)

{ $N_1 = First\_elem(Node\_list)$;

If ($N_1.flag == 0$) $temp = N_1.lba$ else $temp = N_1.lba"$;

If ((($N_1.lbo \leq s - 1$) $\wedge$ ($temp > Vbarcs$) $\wedge$ ($Vbol - Vbshareio = 0$))

$\vee$(($N_1.lbo > s - 1$) $\wedge$ ((($temp > Vbarcs$) $\wedge$ (($N_1.lbo - (s - 1)) \geq (Vbol - Vbshareio)$))))

$\vee$((($N_1.lbo - (s - 1)) > (Vbol - Vbshareio)$) $\wedge$ ($temp \geq Vbarcs$))) *delete* N_1 *from* $Node\_list$;

else

{

If ($N_1.flag == 1$)

{$Child\_list = \emptyset$

$\forall\ v \in N_1.DG.V \mid$ *an unstarred state* $i \in v.states\_cov$

{$Child\_list = Child\_list \cup create\_node(u)$;

$u.freq\_list = N_1.freq\_list$; $u.DG = N_1.DG$; $u.OL = N_1.OL$; $u.lba" = N_1.lba"$;

Star $j \in u.DG.v$; $Star\_list = j$

}

}

else

{$x = First\_elem(N_1.freq\_list)$;$Child\_list = \emptyset$;

$\forall\ PC_x \mid x \in PC_x$,

{$Child\_list = Child\_list \cup create\_node(u)$;$u.PC = PC_x$;}

$\forall\ j \in Child\_list$,

{$j.DG = N_1.DG$, $j.OL = N_1.OL$, $j.freq\_list = N_1.freq\_list$, $j.lba = N_1.lba$;

$j.DG.V = j.DG.V \cup create\_node(u)$; $u.states\_cov = j.PC$;

Star $x \mid x \in u.states\_cov$; $Star\_list = Star\_list \cup x$; *Delete* $u.states\_cov$ *from* $j.freq\_list$;

}

If ($\exists\ j \in Child\_list \mid j.freq\_list == \emptyset$)

{ *delete all* $k \in Child\_list \mid k.freq\_list \neq \emptyset$; *Star any state that occurs only once in* $j.DG.V$;

Add these states to Star_list;

}

}

```
        ∀ j ∈ Child_list,
          {Root_flag = 0; j.DG.E = Porcess_Edges(j.DG); j.OL = Update_output_logic(j.DG);
           j.lbo = Compute_output_logic(j.OL); j.lbs = Compute_states(j.DG);
            If (All states in Freq_list starred and covered)
             {Determine j.total_logic, j.shareio; j.states_used;
                If (j.total_logic = lower bound of cost) return j;
                else if ((j.total_logic ≥ Vblogic) ∨ (j.states_used > Vbstates)) delete j;
                else
                  { Vbarcs = j.arcs; Vblogic = j.total_logic; Vbstates = j.states_used; Vbshareio = j.shareio
                        Vbol = j.output_logic; Best_node = j; Delete j;
                  }
             }
            else if (All states covered but not starred)
                { j.lba'' = | j.DG.E | + Least_additional_arcs(j.DG); j.flag = 1;}
            else {j.lba = Compute_arcs(j.DG); j.flag = 0;}
          }
  Node_list = {Child_list ∪ {Node_list − N1}};
  Order nodes in Node_list by nondecreasing lbs, incase of ties use lba(flag = 0) or lba''(flag = 1) & lbo values;
   }
  }
}
```

The pseudocode for one of the procedures that SOHOT calls is given below

```
Procedure Process_Edges(Node.DG)
{ temp = Star_list;
   ∀ j ∈ temp ∩ {Start_list ∪ End_list},
    { If (j ∈ Start_list) E = E ∪ (H, u | j ∈ u.states_cov);
        else if (j ∈ End_list) E = E ∪ (u | u ∈ u.states_cov, T);
      If ((j ∈ u.states_cov) ∧ (set s ⊆ u.states_cov | s ⊆ Start_list ∪ End_list))
              {Star {s − j}; Delete all k from temp and append k to Star_list | k ∈ {s − j};}
    }
  temp = list of all starred states ∈ u.states_cov | u ∈ Node.DG.V;
  ∀ j ∈ Star_list,
     { If ((j not ∈ end_list) ∧ (j.val + 1 = k.val | k ∈ {temp − j}))
          E=E∪(u1 | u1 ∈ V j ∈ u1.states_cov, u2 | u2 ∈ V j ∈ u2.states_cov);
       else if ((j ∈ end_list) ∧ (∃ k ∈ {temp ∩ Start_list} | (j, k) ∈ Fback_list))
          E=E∪(u1 | u1 ∈ V j ∈ u1.states_cov, u2 | u2 ∈ V j ∈ u2.states_cov);
     }
  If (Root_flag == 0)
     {∀ j ∈ {temp − Star_list},
        { If ((j not ∈ End_list) ∧ (∃ k ∈ Star_list | k.val = j.val + 1))
              E = E ∪ (u1 | j ∈ u1.states_cov, u2 | k ∈ u2.states_cov);
          else if ((j ∈ End_list) ∧ (∃ k ∈ {Star_list ∩ Start_list} | (j, k) ∈ Fback_list))
              E = E ∪ (u1 | j ∈ u1.states_cov, u2 | k ∈ u2.states_cov);
        }
     }
remove multiple arcs if any form E; return E;
}
```

A.2 Other circuits

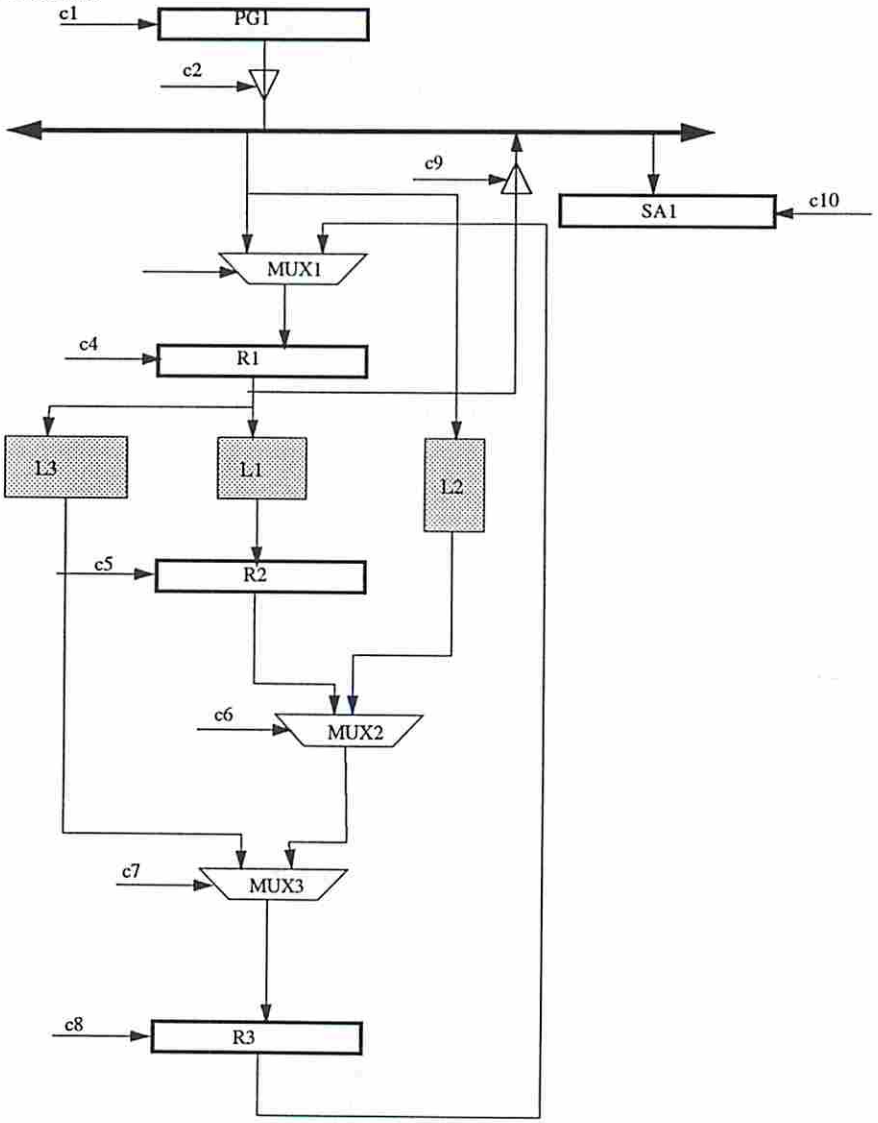


Figure 18: Circuit 2

	TP1	TP2	TP3	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
1	2			1	0	x	x	x	x	x	x	0	0
2	3			0	1	1	1	x	x	x	x	0	0
3	4			0	0	x	x	1	x	x	x	0	0
4	5			0	0	x	x	x	1	0	1	0	0
5	6			0	0	0	1	x	x	x	x	0	0
6	1			0	0	x	x	x	x	x	x	1	1
7		8		1	0	x	x	x	x	x	x	0	0
8		9		0	1	x	x	x	0	0	1	0	0
9		10		0	0	0	1	x	x	x	x	0	0
10		7		0	0	x	x	x	x	x	x	1	1
11			12	1	0	x	x	x	x	x	x	0	0
12			13	0	1	1	1	x	x	x	x	0	0
13			14	0	0	x	x	x	x	1	1	0	0
14			15	0	0	0	1	x	x	x	x	0	0
15			11	0	0	x	x	x	x	x	x	1	1

Figure 19: State transition table for Circuit 2

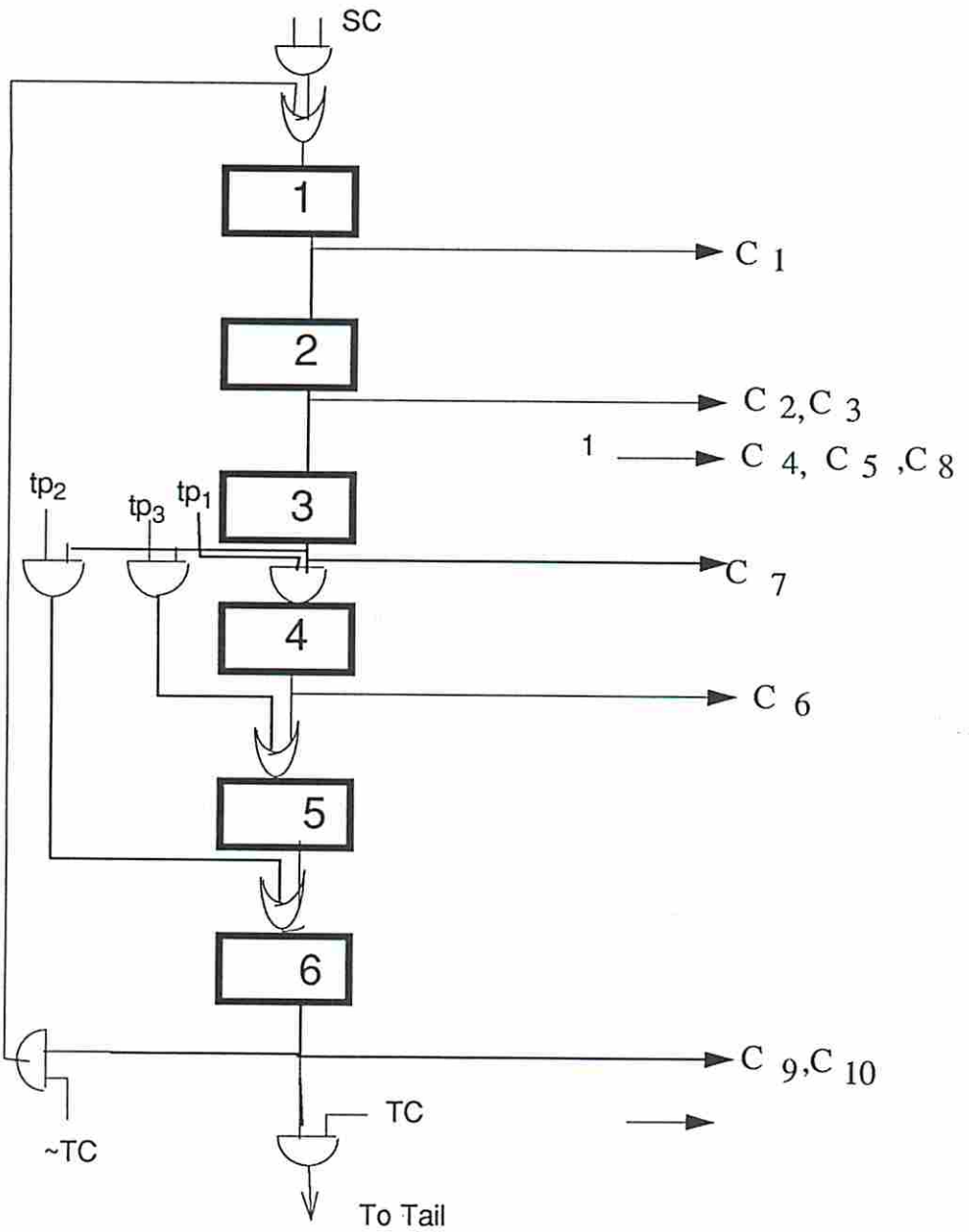


Figure 20: Hardware realization of the minimal merged controller for Circuit 2

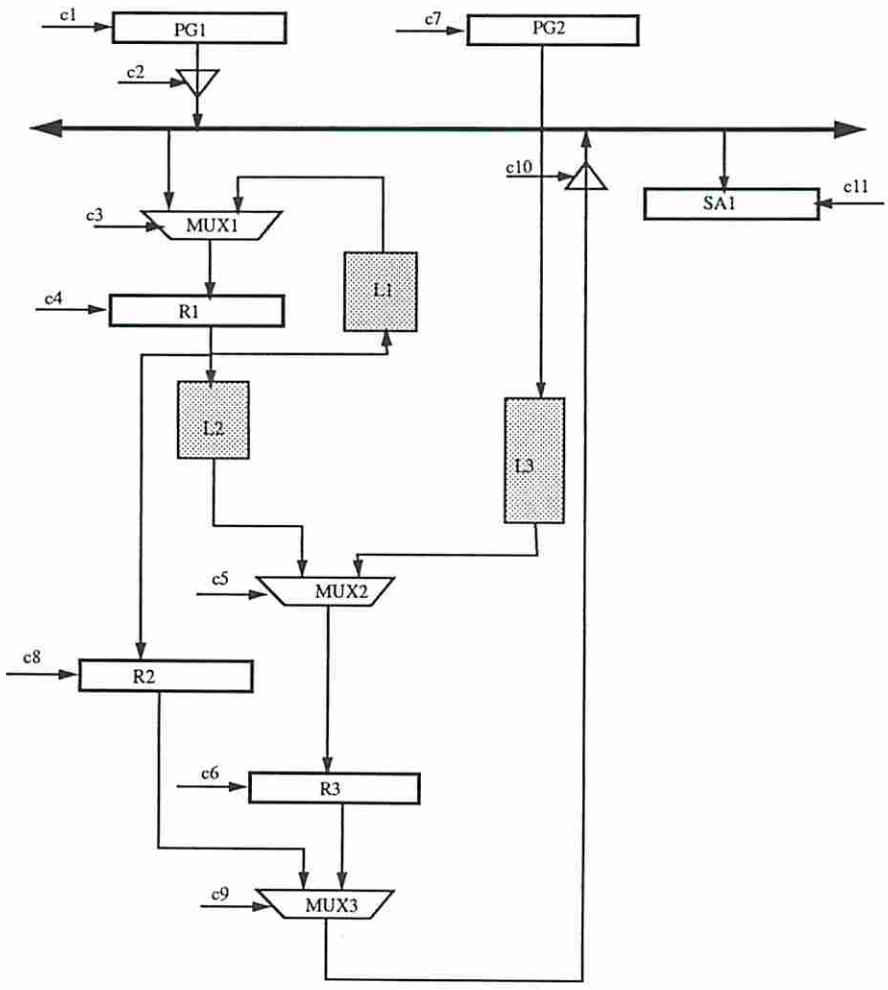


Figure 21: Circuit 3

	tp1	tp2	tp3	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11
1	2			1	0	x	x	x	x	x	x	x	0	0
2	3			0	1	1	1	x	x	x	x	x	0	0
3	4			0	0	0	1	x	x	x	x	x	0	0
4	5			0	0	x	x	x	x	x	1	x	0	0
5	1			0	0	x	x	x	x	x	x	1	1	1
6		7		1	0	x	x	x	x	x	x	x	0	0
7		8		0	1	1	1	x	x	x	x	x	0	0
8		9		0	0	x	x	1	1	x	x	x	0	0
9		6		0	0	x	x	x	x	x	x	0	1	1
10			11	x	0	x	x	x	x	1	x	x	0	0
11			12	x	0	x	x	0	1	0	x	x	0‘	0
12			10	x	0	x	x	x	x	0	x	0	1	1

Figure 22: The transition table for Circuit 3

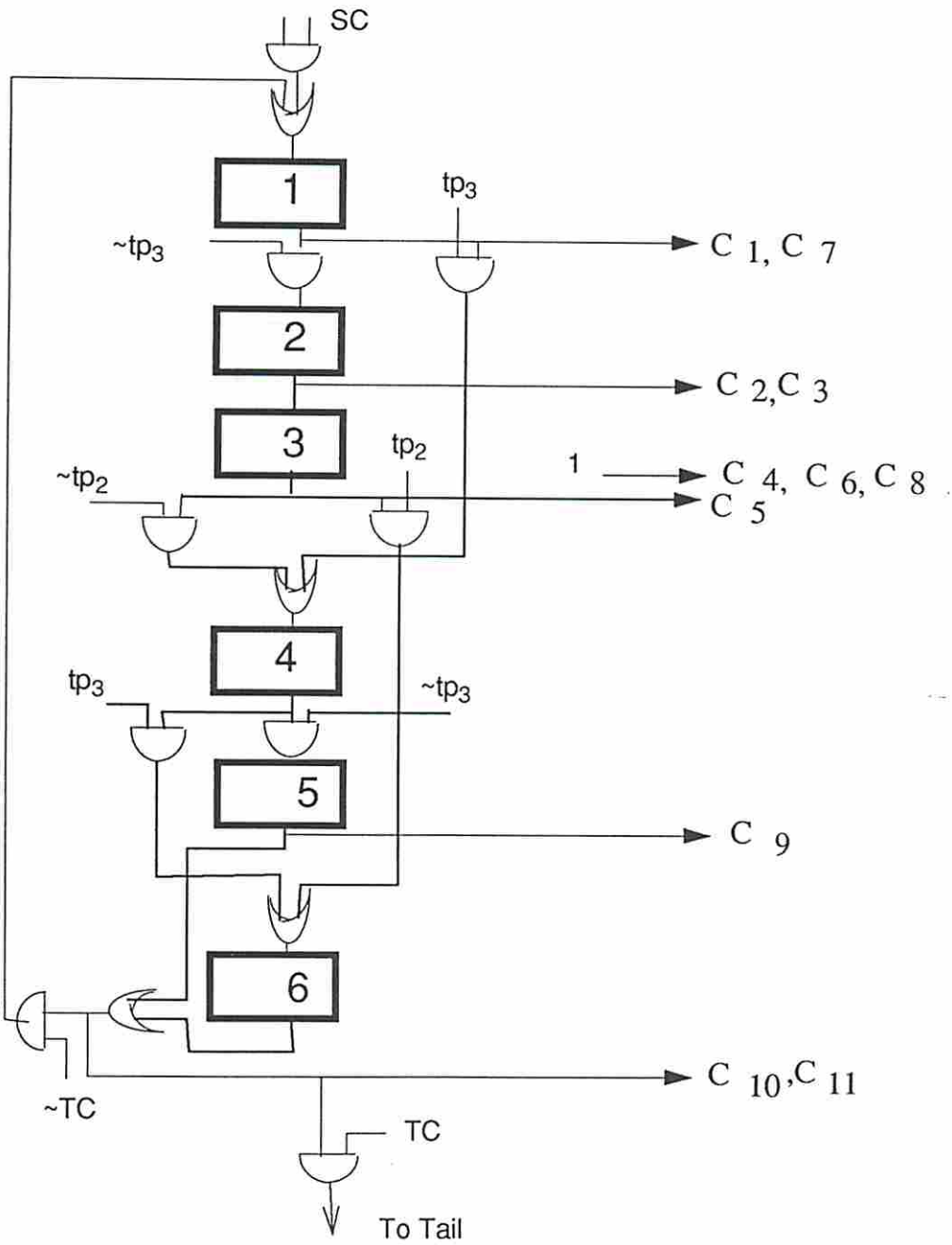


Figure 23: Hardware realization of the minimal merged controller for Circuit 3

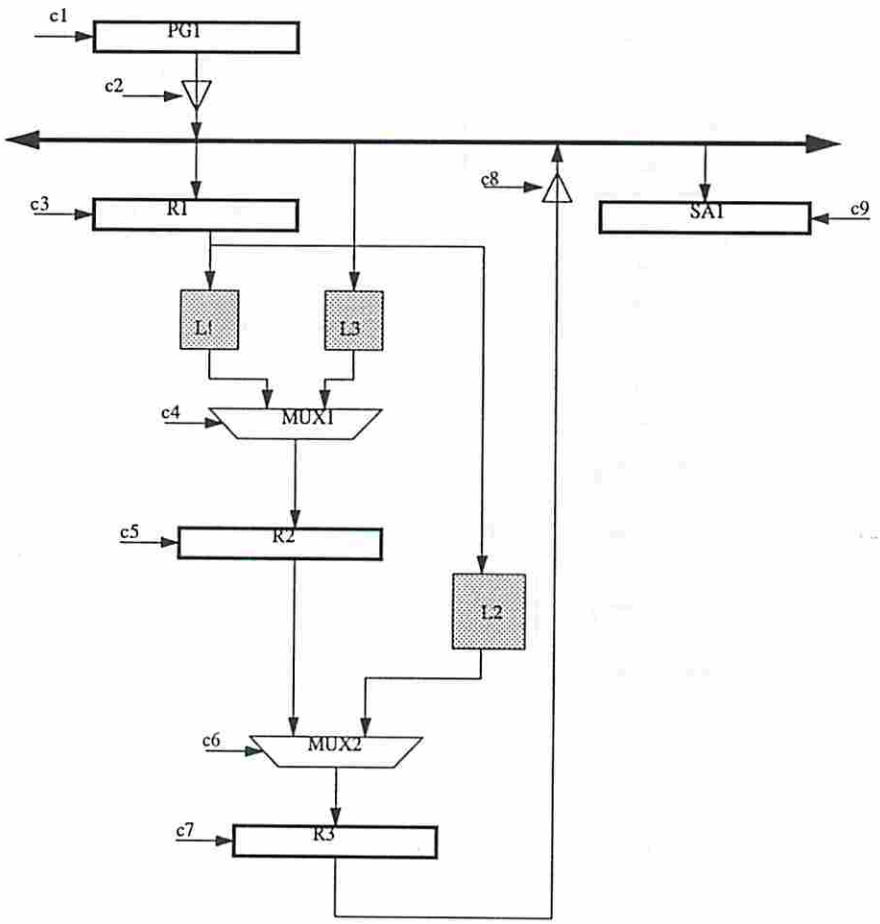


Figure 24: Circuit 4

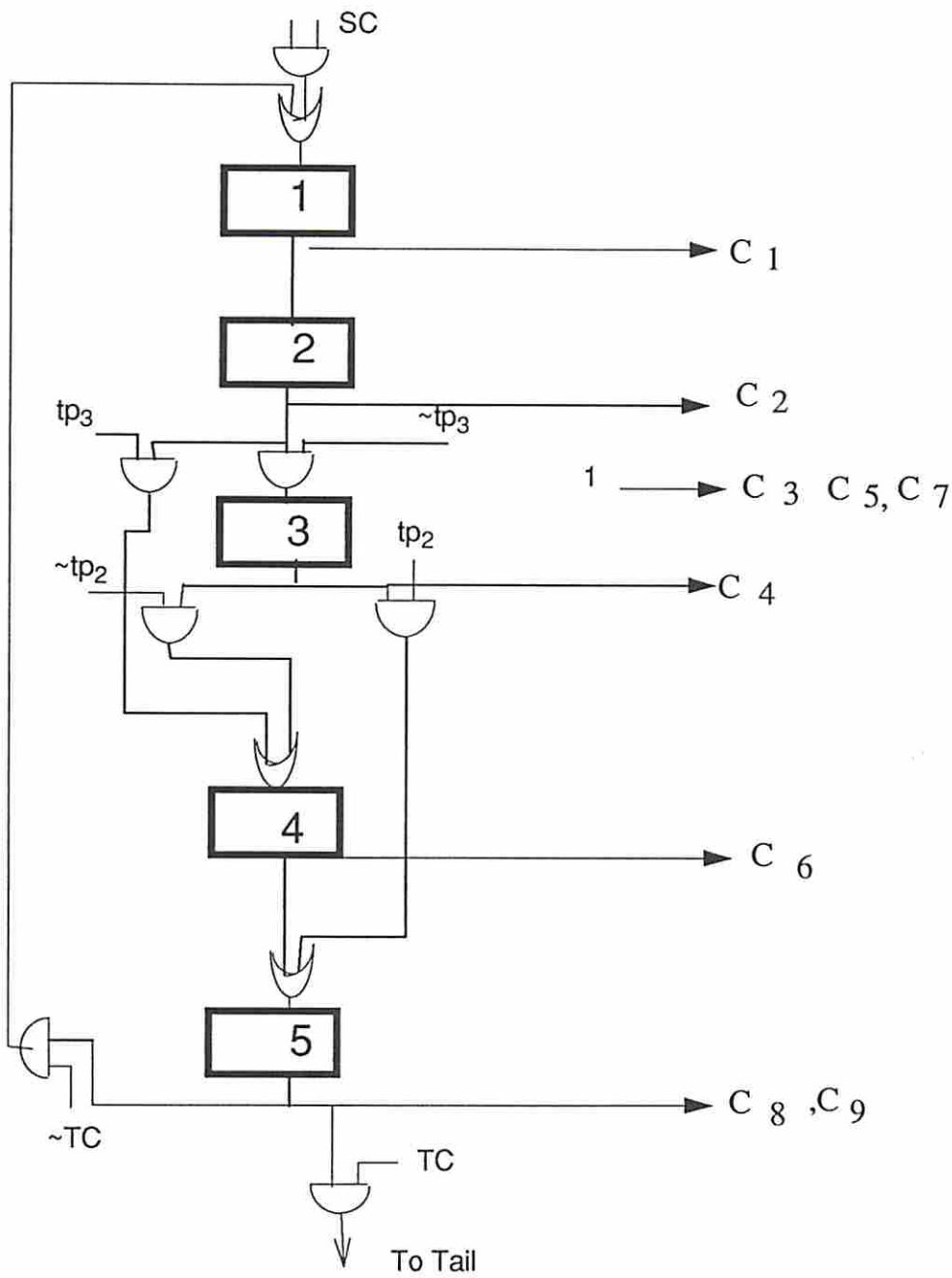


Figure 25: Hardware realization of the minimal merged controller for circuit 4