SIS: A System for Sequential Circuit Synthesis

Electronics Research Laboratory
Memorandum No. UCB/ERL M92/41

Ellen M. Sentovich Kanwar Jit Singh
Luciano Lavagno Cho Moon Rajeev Murgai
Alexander Saldanha Hamid Savoj Paul R. Stephan
Robert K. Brayton Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

4 May 1992

Abstract

SIS is an interactive tool for synthesis and optimization of sequential circuits. Given a state transition table, a signal transition graph, or a logic-level description of a sequential circuit, it produces an optimized net-list in the target technology while preserving the sequential input-output behavior. Many different programs and algorithms have been integrated into SIS, allowing the user to choose among a variety of techniques at each stage of the process. It is built on top of MISII [5] and includes all (combinational) optimization techniques therein as well as many enhancements. SIS serves as both a framework within which various algorithms can be tested and compared, and as a tool for automatic synthesis and optimization of sequential circuits. This paper provides an overview of SIS. The first part contains descriptions of the input specification, STG (state transition graph) manipulation, new logic optimization and verification algorithms, ASTG (asynchronous signal transition graph) manipulation, and synthesis for PGA's (programmable gate arrays). The second part contains a tutorial example illustrating the design process using SIS.

1 Introduction

The SIS synthesis system is specifically targeted for sequential circuits and supports a design methodology that allows the designer to search a larger solution space than was previously possible. In current practice the synthesis of sequential circuits proceeds much like synthesis of combinational circuits: sequential circuits are divided into purely combinational blocks and registers. Combinational optimization techniques are applied to the combinational logic blocks, which are later reconnected to the registers to form a single circuit. This limits the optimization by fixing the register positions and optimizing logic only within combinational blocks without exploiting signal dependencies across register boundaries. Verification techniques are limited to verifying machines with the same encoding. Finally, it is cumbersome to separate the circuit into logic and registers only to reconstruct it later. In this paper, a sequential circuit design methodology is described; it is implemented through a system that employs state-of-the-art synthesis and optimization techniques. This approach is illustrated with an example demonstrating the usefulness of these new techniques and the flexibility the designer can exploit during the synthesis process.

Many algorithms have been published for various stages of sequential synthesis. For synchronous circuits, these include methods for state assignment [24, 58], state minimization [17, 31], testing [16], retiming [22], technology mapping [33], verification [6, 10], timing analysis, and optimization across register boundaries [11, 25, 28, 29]. For asynchronous circuits, these include methods for hazard-free synthesis [20, 32]. However, no comprehensive evaluation of the algorithms and no complete synthesis system in which all of these algorithms are employed has been

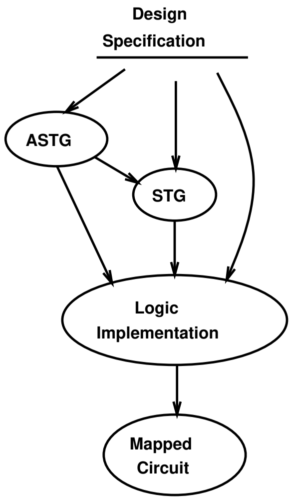


Figure 1: A Design is Specified as an ASTG, STG, or Logic

reported to date. A complete sequential circuit synthesis system is needed, both as a framework for implementing and evaluating new algorithms, and as a tool for automatic synthesis and optimization of sequential circuits.

SIS is an interactive tool like MISII, but for sequential circuit synthesis and optimization. It is built on top of MISII and replaces it in the Octtools [12], the Berkeley synthesis tool set based on the Oct database. While MISII operated on only combinational circuits, SIS handles both combinational and sequential circuits. In the Octtools environment, a behavioral description of combinational logic can be given in a subset of the BDS language (the BDS language was developed at DEC [1]). The program **bdsyn** [45] is used to translate this description into a set of logic equations, and then **bdnet** is used to connect combinational logic and registers and create an Oct description file. Alternately, the starting point can be either a state transition table and SIS is used to invoke state assignment programs to create the initial logic implementation, or a signal transition graph and SIS is used to create a hazard-free logic implementation. SIS is then used for optimization and technology mapping; placement and routing tools in the Octtools produce a symbolic layout for the circuit.

The SIS environment is similar to MISII: optimization is done for area, performance, and testability. System-level timing constraints can be specified for the I/O pins. External "don't care" conditions, expressing given degrees of freedom in the logic equations, can be supplied and used in the optimization. Synthesis proceeds in several phases: state minimization and state assignment, global area minimization and performance optimization, local optimization, and technology mapping. SIS is interactive, but as in MISII scripts are provided to automate the process and guide the optimization steps.

In the sequel, the main new components of SIS will be described (i.e. those algorithms and operations not available in MISII), including an input intermediate format for sequential circuits. This is followed by an example illustrating the use of SIS in the design process.

2 Design Specification

A sequential circuit can be input to SIS in several ways (see Figure 1), allowing SIS to be used at various stages of the design process. The two most common entry points are a net-list of gates and a finite-state machine in state-transition-table form. Other methods of input are by reading from the Oct database, and through a sequential circuit net-list called SLIF (Stanford Logic Interchange Format) [15]. For asynchronous circuits, the input is a signal transition graph [8].

2.1 Logic Implementation (Netlist)

The net-list description is given in extended BLIF (Berkeley Logic Interchange Format) which consists of interconnected single-output combinational gates and latches (see Appendix A for a description). The BLIF format, used in MISII, has been augmented to allow the specification of latches and controlling clocks. The latches are simple generic delay elements; in the technology-mapping phase they are mapped to actual latches in the library. Additionally, the BLIF format accepts user-specified don't care conditions. Designs can be described hierarchically although currently the hierarchy information is not retained in the internal data structure resulting in a flat netlist [1].

2.2 State Transition Graph (STG)

A state transition table for a finite-state machine can be specified with the KISS [26] format, used extensively in state assignment and state minimization programs. Each state is symbolic; the transition table indicates the next symbolic state and output bit-vector given a current state and input bit-vector. External don't care conditions are indicated by a missing transition (i.e., a present-state/input combination that has no specified next-state/output) or by a '-' in an output bit (indicating that for that present-state/input combination that particular output can be either 0 or 1).

2.3 Internal Representation of the Logic and the STG

Internally, the BLIF file for a logic-level specification is represented by two Boolean networks, a **care network**, and a **don't care network** representing the external don't cares. Each network is a DAG, where each node represents either a primary input x_i, a primary output z_i, or an intermediate signal y_i. Each y_i has an associated function F_i, and an edge connects node i to node j if the function at node j depends explicitly on the signal i. The don't care network has the same number of outputs as the care network: an output of '1' in the don't care network under an input condition indicates for that input, either '0' or '1' is allowed for that output in the care network. Simultaneously, the KISS specification (if present) is stored as an STG (state transition graph) structure. The network and STG representations can be given simultaneously by embedding the KISS file in the BLIF specification. SIS provides routines for interactively manipulating both representations of a single circuit as described in Sections 3.1 and 3.2. An example of an STG and a logic implementation is shown in Figure 2; the corresponding BLIF specification with an embedded state table is on the right.

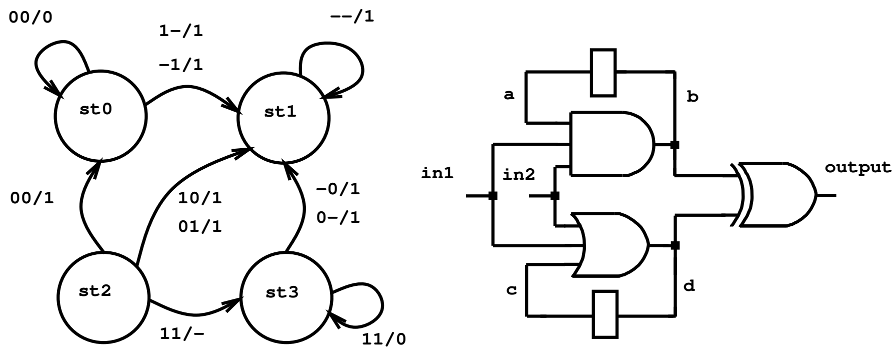
With two internal representations for a single synchronous circuit (STG and logic), it is necessary to check the consistency of the two representations. This is done with the **stg_cover** command, which does a symbolic simulation of the logic implementation for each edge of the STG to ensure that the behavior specified by the edge is implemented by the logic (the STG "covers" the logic implementation). This command should be invoked if the two representations are initially given for a circuit.

2.4 Signal Transition Graph (ASTG)

The signal transition graph is an event-based specification for asynchronous circuits. It is composed of *transitions*, representing changes of values of input or output signals of the specified circuit, and *places*, representing pre– and post–conditions of the transitions. A place can be *marked* with one or more *tokens*, meaning that the corresponding condition holds in the circuit. When all the pre–conditions of a transition are marked, the transition may *fire* (meaning that the corresponding signal changes value), and the tokens are removed from its pre–conditions and added to its post–conditions. Hence a signal transition graph specifies the behavior both of an asynchronous *circuit* and of the *environment* where it operates. The causality relations described by places joining pairs of transitions represent how the circuit and its environment can react to signal transitions.

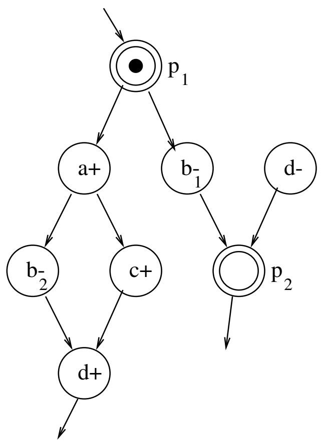
For example, see Figure 3(a), which represents a fragment of a signal transition graph specification. Places are shown as double circles, transitions are shown as simple circles, and places with exactly one predecessor and one successor (*implicit* places) are omitted. Transitions b_1^- and b_2^- are two distinct falling transitions of signal b. Initially, place p_1 is marked. Either a^+ or b_1^- can fire, but not both, since firing one of them removes the token from the

[1] A new program called HSIS (hierarchical SIS) is currently under development for synthesis of hierarchical netlists.




```
.model circuit
.inputs in1 in2
.outputs output
.start_kiss
.i 2
.o 1
00 st0 st0 0
-1 st0 st1 1
...
.end_kiss
.latch b a
.latch d c
.names a in1 in2 b
111 1
.names c in1 in2 d
--1 1
-1- 1
1-- 1
.names b d output
01 1
10 1
.end
```

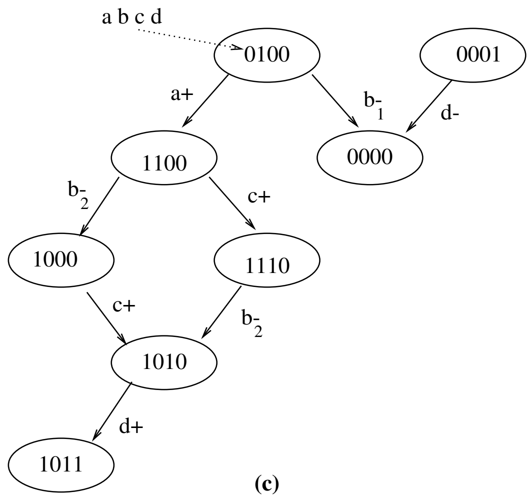
Figure 2: STG, Logic Implementation, Partial BLIF file



(a)

```
.model example
.inputs a b
.outputs c d
.graph
p1 a+ b-/1
a+ b-/2 c+
b-/2 d+
c+ d+
b-/1 p2
d- p2
....
.marking { p1 }
.end
```

(b)



(c)

Figure 3: A Signal Transition Graph, its text file representation and its State Graph

predecessor of the other one (a^+ and b_1^- are *in conflict*). If a^+ fires, signal a changes value from 0 to 1. If b_1^- fires, signal b changes value from 1 to 0. Standard digital circuits are *deterministic*, so this kind of non-deterministic *choice* between firing transition is allowed only for *input* signals (it allows a more abstract modeling of the circuit environment). If a^+ fires, it marks the implicit places between a^+ and b_2^- and between a^+ and c^+. So a^+ *causes* transitions b_2^- and c^+ to be enabled (a^+ and b_2^- are *ordered*). Transitions b_2^- and c^+ do not share a pre–condition, so they can fire in any order (b_2^- and c^+ are *concurrent*). When *both* have fired, d^+ becomes enabled, because both its pre–conditions are marked, and so on.

Note that the transitions in this fragment allow us to *infer* a value for each signal in each distinct marking of the signal transition graph. For example, in the initial marking $a = 0$ (because its first transition, a^+, will be rising), $b = 1$ (because its first transition, either b_1^- or b_2^-, will be falling), and similarly $c = 0$ and $d = 0$. The value label attached to each marking must be *consistent* across different firing sequences of the signal transition graph. For example, when p_2 is marked, the value for d is always 0, independent of whether we fired d^- or b_1^- (in the latter case the value for d is the same as when p_1 is marked, since no transition of d fires between p_1 and p_2).

The existence of such a consistent labeling of each marking with signal values is the key for the *correctness* of a signal transition graph, together with three other properties:

- *liveness*: for every transition, from every marking that can be reached from the initial one, we must be able to reach a marking where the transition is enabled (no deadlocks and no "operating cycles" that cannot be reached from each other).
- *safeness*: no more than one token can be present in a place in any reachable marking.
- *free-choice*: if a place has more than one successor (p_1 in the example), then it must be the *only predecessor* of those transitions.

Furthermore a signal transition graph must be *pure*, i.e. no place can be both a predecessor and a successor of the same transition, and *place-simple*, i.e. no two places can have exactly the same predecessors and successors.

A *state machine* (SM) is a signal transition graph where each transition has exactly one predecessor and one successor place (no concurrency). A *marked graph* (MG) is a signal transition graph where each place has exactly one predecessor and one successor transition (no choice). These two classes of signal transition graphs are useful because some synthesis algorithms can be used only, for example, on marked graphs, and some analysis algorithms decompose a general signal transition graph into state machine components or into marked graph components (see Section 3.3).

A signal transition graph can be represented as directed graph in a text file and read into SIS with the **read_astg** command (an example is shown in Figure 3(b)). The first three lines (`.model`, `.inputs` and `.outputs`) have the same meaning as in the BLIF format. In addition, the keyword `.internal` can be used to describe signals that are not visible to the circuit environment, e.g. state signals. Everything after a "#" sign is treated as a comment.

Each line after `.graph` describes a set of graph edges. For example `p1 a+ b-/1` describes the pair of edges between p_1 and a^+ and b_1^- respectively. The optional `.marking` line describes the set of initially marked places (a blank-separated list surrounded by braces). Implicit places, e.g. the place between a^+ and b_2^-, can be denoted in the initial marking by writing its predecessor and successor transitions between angle brackets, e.g. `<a+,b-/2>`.

In addition to input, output and internal transitions, the signal transition graph model also allows *dummy* signals. Dummy signals, denoted by the keyword `.dummy`, just act as "placeholders" or "synchronizers" and do not represent any actual circuit signal (so their transitions are not allowed to have a "+" or "-" sign). Dummy signals are very useful, for example, in transforming an asynchronous finite state machine into a signal transition graph by direct translation [9]. In this translation one explicit place represents each state, and it has one dummy transition for each output edge, followed by a set of transitions representing input signal changes[2], followed by state signal changes, followed by output signal changes, followed by another dummy transition, followed by the place corresponding to the next state. Note that the input transitions must precede the state and output transitions in order to imply the correct deterministic behavior, because the dummy transitions have no effect on the observable signals. In order for this direct translation to be possible, every finite state machine state must be entered with a single set of values of input and output signals,

[2] If more than one input signal changes value, this situation could not be represented, in general, by a *free-choice* signal transition graph without dummy transitions.

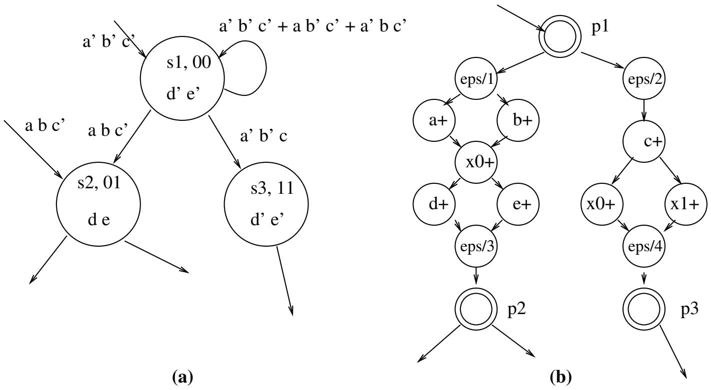


Figure 4: An asynchronous finite state machine and its signal transition graph specification

i.e. it must be a Moore-type machine, and each edge entering the state must have exactly the same input signal labels (this is often the case, for example, with classical "primitive flow tables", see [55]).

Suppose that a finite state machine, whose fragment appears in Figure 4(a), has a state s_1 with code 00, where all input and output signals signals have the value 0. If inputs a and b rise, then it makes a transition to a state s_2, with code 01, where outputs d and e have value 1. Otherwise, if input c rises, it makes a transition to state s_3, with code 11, where no output change occurs.

The finite state machine fragment can be represented by the following signal transition graph fragment (place `p1` corresponds to state s_1, place `p2` to state s_2, and place `p3` to state s_3), shown in Figure 4(b):

```
.inputs a b c ...
.outputs d e ...
# state signals
.internal x0 x1
.dummy eps
.graph
...

# state s1: two transitions
p1 eps/1 eps/2

# input change
eps/1 a+ b+
# state change 00 -> 01
a+ x0+
b+ x0+
# output change
x0+ d+ e+
# next state is s2
d+ eps/3
e+ eps/3
eps/3 p2

# input change
eps/2 c+
# state change 00 -> 11
c+ x0+ x1+
```

```
# no output change
# next state is s3
x0+ eps/4
x1+ eps/4
eps/4 p3

...
```

3 Part I: The SIS Synthesis and Optimization System

SIS contains many new operations and algorithms, and it is the combination of these within a uniform framework that allows the designer to explore a large design space. Some of these are enhancements to the combinational techniques previously employed in MISII. These include improvements to performance optimization (both restructuring and technology mapping), storage and use of external don't cares, improved node minimization, and faster divisor extraction. In addition, new sequential techniques are included: an interface to state minimization and state assignment programs, retiming, sequential circuit optimization algorithms, finite-state machine verification, technology mapping for sequential circuits, and synthesis of asynchronous designs.

3.1 STG Manipulations

3.1.1 State Minimization

A state transition graph (STG) contains a collection of symbolic states and transitions between them. The degrees of freedom in this representation, or don't cares, are either unspecified transitions or explicit output don't cares. These degrees of freedom, along with the notion that two states are equivalent if they produce equivalent output sequences given equivalent input sequences, can be exploited to produce a machine with fewer states. This usually translates to a smaller logic implementation. Such a technique is called **state minimization** and has been studied extensively (e.g. [17, 31]). For completely specified machines the problem can be solved in polynomial time, while in the more general case of incompletely specified machines the problem is NP hard.

In SIS, state minimization has been implemented to allow the user to choose among various state minimization programs. No single state minimization algorithm has been integrated, but rather, the user can choose among state minimization programs, distributed with SIS, or other programs that conform to an I/O specification designed for state minimization programs. Programs that conform to the specification can be executed from the SIS shell. It has simple requirements, e.g. that the program use the KISS format for input and output. For example, the user may invoke the STAMINA [17] program, a heuristic state minimizer for incompletely specified machines distributed with SIS, as follows:

```
sis> state_minimize stamina -s 1
```

In this case, STAMINA is used with its option '-s 1' to perform state minimization on the current STG. When the command is completed, the original STG is replaced by the one computed by STAMINA with (possibly) fewer states.

3.1.2 State Assignment

State assignment provides the mapping from an STG to a netlist. State assignment programs start with a state transition table and compute optimal binary codes for each symbolic state. These binary codes are used to create a logic-level implementation by substituting the binary codes for the symbolic states and creating a latch for each bit of the binary code.

In SIS, the state assignment mechanism is similar to state minimization: the user is free to choose among state assignment programs, provided those programs conform to a simple specification (in this case, the input is KISS and the output is BLIF). A call to a state assignment program, such as

```
sis> state_assign nova -e ih
```

will perform optimal state assignment on the STG and return a corresponding logic implementation. Two state assignment programs, JEDI[24] and NOVA [58] are distributed with SIS. JEDI is a general symbolic encoding program (i.e., for encoding both inputs and outputs) that can be used for the more specific state encoding problem; it is targeted for multi-level implementations. NOVA is a state assignment program targeted for PLA-based finite-state machines; it produces good results for multi-level implementations as well.

After state assignment there may be unused state codes, and from the KISS specification, unspecified transitions and output don't cares. These are external don't cares for the resulting logic, and can be specified in the BLIF file that is returned by the state assignment program, as is done with NOVA. These don't care conditions will be stored and automatically used in some subsequent optimization algorithms, such as **full_simplify** (see Section 3.2.1).

3.1.3 STG Extraction

STG extraction is the inverse of state assignment. Given a logic-level implementation, the state transition graph can be extracted from the logic for subsequent state minimization or state assignment. This provides a way of re-encoding the machine for a possibly more efficient implementation. The STG extraction algorithm involves forward simulation and backtracking to find all states reachable from the initial state. Alternately, *all* 2^n states can be extracted, where n is the number of latches. The resulting STG is completely specified because it is derived from the implementation (which is completely specified by definition) and because the external don't cares in the don't care network are not used during the extraction. Thus to minimize the size of the STG the don't cares should be given explicitly for the STG representation. The **stg_extract** operation should be used with caution as the STG may be too large to extract; the STG is in effect a two-level (symbolic) sum-of-products form.

3.2 Netlist Manipulations

SIS contains MISII and all the combinational optimization techniques therein. This includes the global area minimization strategy (iteratively extract and resubstitute common sub-expressions, eliminate the least useful factors), combinational speed-up techniques based on local restructuring, local optimization (node factoring and decomposition, two-level logic minimization using local don't care conditions), and technology mapping for area or performance optimization. Some of these techniques have been improved significantly over the last available version of MISII. In addition, new techniques for both combinational and sequential circuits have been introduced.

3.2.1 Combinational Optimization

Node Simplification

The logic function at a node is simplified in MISII using the **simplify** command which uses the two-level logic minimizer ESPRESSO [4]. The objective of a general two-level logic minimizer is to find a logic representation with a minimal number of implicants and literals while preserving the functionality. There are several approaches to this problem. In ESPRESSO, the offset is generated to determine whether a given cube is an implicant in the function. The input usually contains a cover for the onset and a cover for the don't care set. A cover for the offset is generated from the input using a complement algorithm based on the Unate Recursive Paradigm [4]. The number of cubes in the offset can grow exponentially with the number of input variables; hence the offset generation could be quite time consuming. In the context of a multi-level network, node functions with many cubes in the offset and don't care set happen quite often, but the initial cover of the onset is usually small, and both the initial cover and the don't care cover mainly consist of primes.

To overcome the problem of generating huge offsets, a subset of the offset called the reduced offset is used [27]. The reduced offset for a cube is never larger than the entire offset of the function and in practice has been found to be much smaller. The reduced offset can be used in the same way as the full offset for the expansion of a cube and no quality is lost. The use of the reduced offset speeds up the node simplification; however, if the size of don't care set is too large, the computation of reduced offsets is not possible either. As a result, filters [40] must be introduced to keep the don't care size reasonably small. The filters are chosen heuristically with the goal that the quality of the node simplification is not reduced significantly.

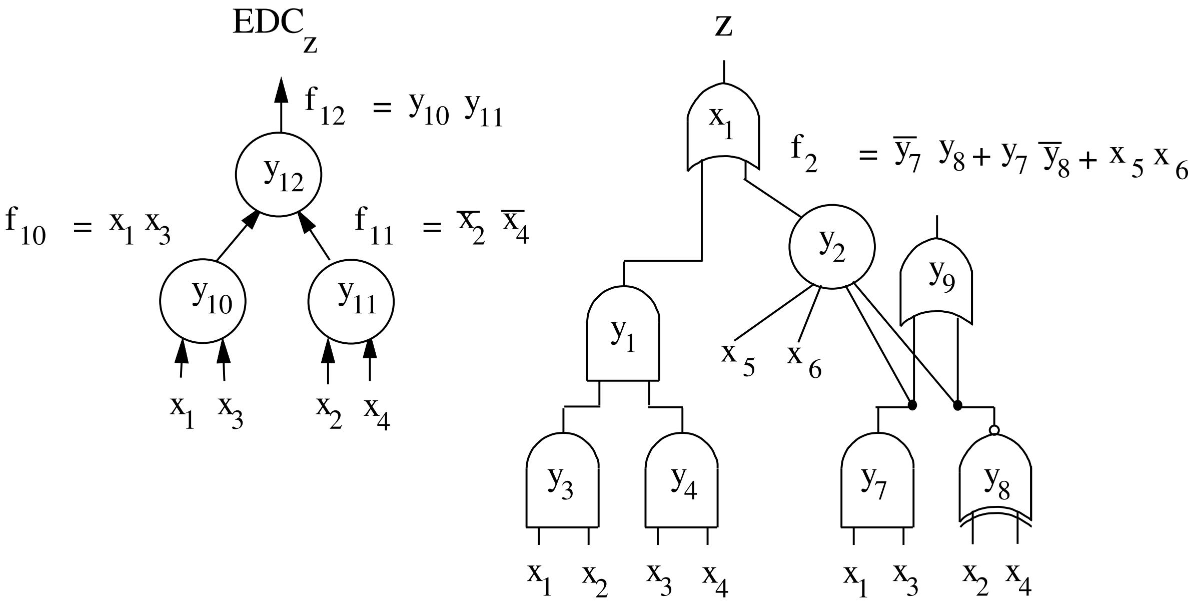


Figure 5: Node Simplification

To perform node simplification on a node in a multi-level network, an appropriate don't care set must first be generated. In MISII, subsets of the satisfiability and observability don't care sets (SDC and ODC respectively, see [2]) are used. The SDC captures conditions which can never happen in the network and hence are don't cares (i.e. if $y_1 = \bar{x}_1$, then $y_1 x_1$ is a don't care because y_1 and x_1 will never both be 1). The ODC captures changes which may occur without affecting the outputs (i.e. if y_1 is the only output and $y_1 = y_2 y_3$, then $\bar{y}_2$ is a don't care for y_3 because when $y_2 = 0$ the value of y_3 is not observable at y_1).

In latter versions of MISII and in SIS, the generation of the don't care sets has been improved significantly. A subset of the SDC is used that is known as the support subset [40] and consists of the satisfiability don't cares of all the nodes whose support is included in the support of the node being simplified. By using this subset we can effectively cause a Boolean substitution of the nodes of the network into the node being simplified, but in general we do not get maximum simplification of the node. As an example of the subset filter, while simplifying node y_2 shown in Figure 5, the SDC for node y_9 is generated because the support of y_9 is a subset of the support of y_2. Thus substitution of node y_9 in y_2 will happen if y_2 can be expressed in terms of y_9 and it results in a simpler function for y_2.

The techniques for computing observability don't cares compute these in terms of intermediate variables in the network. External don't cares (EDC) are expressed with a separate don't care network, as described in Section 2. To fully utilize ODC's plus EDC's for the simplification of each intermediate node one has to find how the current representation of the node is related to these don't cares. The relation between EDC's plus ODC's and the current representation at each node is usually only through primary inputs. To get the most simplification possible for each node, one has to provide this connection, which is the structure of the Boolean network, to the two-level minimizer.

The most straightforward approach is to establish this connection through SDC's. SDC's are generated for all the nodes in the transitive fanin cone of the node being simplified to relate the current representation of the node to the primary inputs. SDC's are also generated to relate the EDC plus ODC to the primary inputs. These are all the nodes in the transitive fanin cone of the support of the EDC plus ODC.

Example: The EDC plus ODC for y_2 in Figure 5 is $d_2 = y_1 + y_{12}$ (y_1 is from the ODC and y_{12} is from the EDC). SDC's for nodes y_7 and y_8 relate y_2 to primary inputs. SDC's for nodes $y_1, y_3, y_4, y_{10}, y_{11}$, and y_{12} relate d_2 to primary inputs. We also generate SDC of y_9 because it may be substituted in the representation of y_2. The input to the two-level minimizer has 15 input variables and 33 cubes for this very small example. After node simplification, the representation at y_2 becomes $f_2 = y_9 + x_5 x_6$.

It is obvious that this approach is not practical for networks with many levels; the size of satisfiability don't care set grows very large in such cases and node simplification becomes computationally too expensive.

To reduce the size of the input to the two-level minimizer, several techniques are employed. External and observability don't cares are computed for each node using the techniques in [41]. The external don't cares are only allowed in two-level form expressed directly in terms of primary inputs. A subset of the ODC called the CODC (compatible ODC) is computed for the simplification of each node. The compatible property ensures that the don't care sets on nodes can be used simultaneously with no need for re-computation after each node has been simplified. CODC's are computed in terms of intermediate variables in the network. A collapsing and filtering procedure is used to find a subset of the CODC which is in the transitive fanin cone of the node being simplified. A limited SDC is generated to use the CODC plus EDC in two-level form. However, EDC's cannot be represented in two-level form in many cases because the number of cubes in the sum-of-products representation of EDC's grows very large. Also, because of collapsing and filtering and the limited SDC generated the quality is reduced considerably compared to what could be obtained using the full don't care set.

Example: The EDC plus ODC for y_2 in Figure 5 in terms of primary inputs is $d_2 = x_1x_2x_3x_4 + x_1\overline{x}_2x_3\overline{x}_4$. SDC's for nodes y_7, y_8 and y_9 must be generated. The input to the two-level minimizer has 9 input variables and 15 cubes. After node simplification, the representation at y_2 becomes as before $f_2 = y_9 + x_5x_6$.

There is one final improvement to the don't care set computation. Since each node has a local function $f_i : B^r \longrightarrow B$ (assuming r fanins), ideally one would like to express the external plus observability don't cares of each node in terms of its immediate fanins, not the primary inputs. These are minterms $m_l \in B^r$ for which the value of f_i can be either 1 or 0 and the behavior of the Boolean network is unchanged. These local don't cares are related to the EDC, ODC of y_i, and SDC's of the network and are as effective in node simplification as the full don't care set.

Let $y_\circ$ be the node being simplified and $f_\circ : B^r \longrightarrow B$ be the local function at this node in terms of its fanins $y_1, \ldots, y_r$. The local don't care set $d^l_\circ$ is all the points in B^r for which the value of $f_\circ$ is not important.

To find $d^l_\circ$, we first find the observability plus external don't care set, $d^g_\circ$, in terms of primary inputs. The care set of $y_\circ$ in terms of primary inputs is $\overline{d}^g_\circ$. The local care set $\overline{d^l_\circ}$ is computed by finding all combinations in B^r reachable from $\overline{d}^g_\circ$. Any combination in B^r that is not reachable from $\overline{d}^g_\circ$ is in the local don't care set $d^l_\circ$.

Example: The local don't care for y_2 is $d^{lm}_2 = y_7y_8$ (computed by effectively simulating the function $x_1x_2x_3x_4 + x_1\bar{x}_2x_3\bar{x}_4$ from the previous example). By examining the subset support, we determine that the SDC of y_9 should be included. The input to the two-level minimizer has 5 inputs and 7 cubes. After node simplification, the representation at y_2 becomes as before $f_2 = y_9 + x_5x_6$.

The command **full_simplify** in SIS uses the satisfiability, observability, and external don't cares to optimize each node in a multi-level network as described above. It first computes a set of external plus observability don't cares for each node: a depth-first traversal of the network starts at the outputs with the external don't cares, and works backward computing CODC's for each node. At each intermediate node, the local don't cares are computed in terms of fanins of the node being simplified by an image computation which uses BDD's. This don't care set is augmented with some local SDC's, and minimized with the two-level minimizer ESPRESSO [4]. Results for **full_simplify** are reported in [42]. This command should not be applied to networks where the BDD's become too large; there is a built-in mechanism to halt the computation if this occurs.

Kernel and Cube Extraction

An important step in network optimization is extracting new nodes representing logic functions that are factors of other nodes. We do not know of any Boolean decomposition technique that performs well and is not computationally expensive; therefore we use algebraic techniques. The basic idea is to look for expressions that are observed many times in the nodes of the network and extract such expressions. The extracted expression is implemented only once and the output of that node replaces the expression in any other node in the network where the expression appears. This technique is dependent on the sum-of-products representation at each node in the network and therefore a slight change at a node can cause a large change in the final result, for better or for worse.

The current algebraic techniques in MISII are based on kernels [3]. The kernels of a logic expression f are defined as

$$K(f) = \{g \mid g = f/c, g \ is \ cube \ free \ \}$$

where c is a cube, g has at least two cubes and is the result of algebraic division of f by c, and there is no common literal among all the cubes in g (i.e. g is cube free). This set is smaller than the set of all algebraic divisors of the nodes in the network, therefore it can be computed much faster and is almost as effective. One problem encountered with this in practice is that the number kernels of a logic expression is exponentially larger than the number of cubes appearing in that expression. Furthermore, after a kernel is extracted, there is no easy way of updating other kernels, so kernel extraction is usually repeated. Once all kernels are extracted the largest intersection that divides most nodes in the network is sought. There is no notion of the complement of a kernel being used at this stage. After kernels are extracted, one looks for the best single cube divisors and extracts such cubes. The kernels and cubes are sought only in normal form. Later on, Boolean or algebraic resubstitution can affect division by the complement as well. Extraction of multi-cube divisors and common cubes from a network in MISII is done with the **gkx** and **gcx** commands.

A more recent algebraic technique extracts only two-cube divisors and two-literal single-cube divisors both in normal and complement form [57]. This approach has several advantages in terms of computation time while the quality of the final result is as good as kernel-based approaches according to [57] as well as our experimental results. It is shown that the total number of double-cube divisors and two-literal single-cube divisors is polynomial in the number of cubes appearing in the expression. Also, this set is created once, and can be efficiently updated when a divisor is extracted. Additionally, one looks for both normal and complement form of a divisor in all the nodes in the network so in choosing the best divisor a better evaluation can be made based on the usefulness of the divisor as well as its complement. There is also no need for algebraic resubstitution once divisors are extracted.

The algorithm works as follows. First all two-cube divisors and two-literal single-cube divisors are recognized and put in a list. A weight is associated with each divisor which measures how many literals are saved if that expression is extracted. This weight includes the usefulness of the complement in the cases where the complements are single cubes or other two-cube divisors. Common cube divisors are also evaluated at the same time so that "kernel" extraction and "cube" extraction are nicely interleaved by this process. The divisor with highest weight is extracted greedily. All other divisors and their weights are updated and the whole process is repeated until no more divisors can be extracted. This technique has been implemented in SIS and is called **fast_extract** or **fx**. Generally, it should replace the MISII commands **gkx** and **gcx** since its quality is comparable but its speed in many cases is substantially higher [43].

One shortcoming of this approach is that the size of each divisor is limited to no more than two cubes. However, large nodes are effectively extracted by the combined process of **fast_extract** and elimination. Elimination is used to increase the size of some divisors and remove others that are not effective.

Test Pattern Generation and Redundancy Removal

A very efficient automatic test pattern generation algorithm for combinational logic has been implemented in SIS. First, fault collapsing is performed across simple gates; both fault equivalence and fault dominance are used to reduce the fault list. Random test generation is done using parallel fault simulation. After the random patterns have been simulated, the algorithm performs a deterministic search to find tests for the remaining faults. This part is based on the algorithm reported in [19]: a set of equations is written to express the difference between the good and faulty circuits for a particular fault and Boolean satisfiability is used to find a satisfying assignment for these equations. The implementation of the algorithm in SIS has been substantially improved for speed. While the single-stuck-fault test pattern generation problem is NP-complete, this implementation has been able to produce 100% stuck-at fault coverage in all test cases available to us. The command **atpg file** in SIS produces a set of test vectors in the file **file**. The redundancy removal command **red_removal** is based on these techniques and iteratively removes all redundant faults.

Technology Mapping

MISII uses a tree-covering algorithm to map arbitrary complex logic gates into gates specified in a technology library [39]. (The technology library is given in **genlib** format, which is described in Appendix B.) This is done by decomposing the logic to be mapped into a network of 2-input NAND gates and inverters. This network is then “covered” by patterns that represent the possible choices of gates in the library. During the covering stage the area or the delay of the circuit is used as an optimization criterion. This has been successful for area minimization but ineffective for performance optimization because the load at the output of gates is not considered when selecting gates and no optimization is provided for signals that are distributed to several other gates. Recent refinements have significantly improved it for performance optimization, as reported in [54]; tree covering has been extended to take load values into account accurately. In addition, efficient heuristic fanout optimization algorithms, reported in [51], have been extended and implemented. They are also used during tree covering to estimate delay at multiple fanout points; this enables the algorithm to make better decisions in successive mapping passes over the circuit.

In addition to the technology mapping for delay another command that inserts buffers in the circuit to reduce delay, is provided. The command **buffer_opt** [47] takes as input a circuit mapped for minimum area (`map -m 0`) and inserts buffers to improve circuit performance. This command is subsumed by the `map -m 1 -A` command which in addition to adding buffers also makes selections of the gates to implement the logic functions. However, it does provide a design whose area and delay are intermediate to the minimum-area and minimum-delay mapped circuits.

Restructuring for Performance

The delay of the circuit is determined by the multi-level structure of the logic as well as the choice of gates used to implement the functions. There are two algorithms that address the restructuring of the logic to improve the delay. These are **speed_up** [48] and **reduce_depth** [52].

The **speed_up** command operates on a decomposition of the network in terms of simple gates (2-input NAND gates and inverters). This description is the same as that used by technology mapping. The speed_up command tries to reduce the depth of the decomposed circuit with the intuition that a small depth representation will result in a smaller delay. The restructuring is performed by collapsing sections along the long paths and resynthesizing them for better timing characteristics [48]. Following the restructuring the circuit is mapped for minimum delay. The recommended use of the speed_up command is to run the following on an area-optimized circuit: `gd *; eliminate -1; speed_up; map -m 1 -A`.

The **reduce_depth** command takes a different approach to technology-independent network restructuring for performance optimization [52]. The idea is to uniformly reduce the depth of the circuit. It does this by first clustering nodes according to some criteria and then collapsing each cluster into a single node. The clusters are formed as follows: a maximum cluster size is computed, and the algorithm finds a clustering that respects this size limit and minimizes the number of levels in the network after collapsing the clusters. The cluster size limit can be controlled by the user. The clustering and collapsing results in a large area increase. Hence, it is important to follow the clustering with constrained area-optimization techniques that will not increase the circuit depth. This is done by using the `simplify -l` and `fx -l` commands. The `-l` option ensures that depth of the circuit will not increase. A special script **script.delay** that uses the **reduce_depth** command is provided for automating synthesis for performance optimization.

3.2.2 Sequential Optimization

All of the combinational techniques described above can be applied to the combinational blocks between register boundaries in a sequential circuit. Further improvements can be made by allowing the registers to move and by performing optimizations across register boundaries. Some of these techniques are described in the following section, including retiming, retiming and resynthesis, and retiming don't cares. Some extensions are required for the technology mapping algorithms to work on sequential circuits. Finally, sequential don't cares based on unreachable states can be computed and used during node minimization.

Retiming

Retiming [22] is an algorithm that moves registers across logic gates either to minimize cycle time, minimize the number of registers, or minimize the number of registers subject to a cycle-time constraint. It operates on synchronous edge-triggered designs. A particular gate is retimed forward by moving registers from each of its fanins to each of its fanouts (see Figure 6). The sequential I/O behavior of the circuit is maintained. An example of retiming for minimum cycle time taken from [21] is shown in Figure 7. This example represents a correlator circuit, which takes a stream of bits and compares it with a pattern a_0, a_1, a_2, a_3. The delays of each of the components are given (delay = 3 for a comparator, 7 for an adder). For this circuit, the cycle time is reduced from 24 to 13.

The original algorithm [23] was based on a mixed-integer linear programming formulation for determining whether a particular cycle time is feasible; if so, the correct register positions are determined. Later, more efficient relaxation-

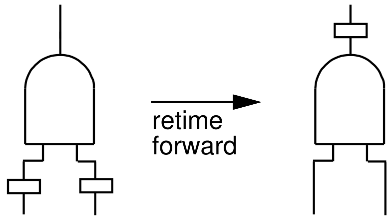


Figure 6: Retiming a Gate Forward

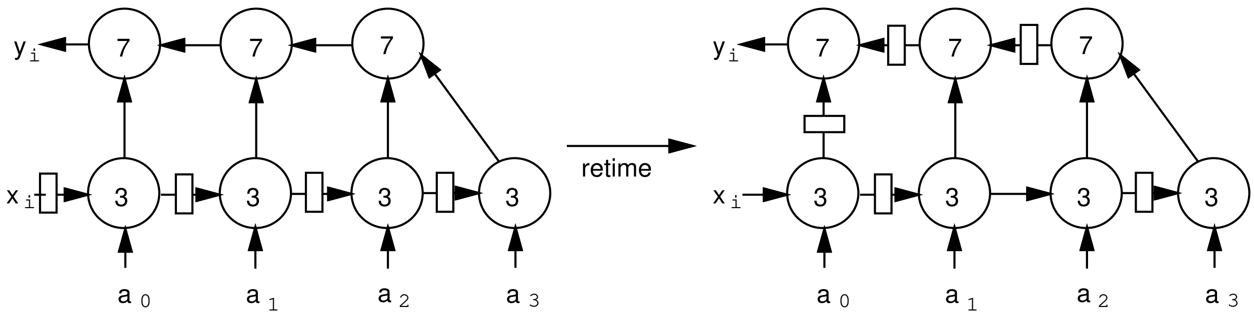


Figure 7: Retiming the Correlator Circuit

based techniques were reported [44]. Both approaches are implemented in the **retime** command in SIS. To determine the optimal cycle time, the feasible cycle time algorithm is combined with a binary search procedure through a range of cycle times.

The original circuit may have an initial state associated with the registers, and this initial state should be preserved during retiming as it may affect subsequent optimization and behavior of the circuit. Although the retimed circuit may not have an equivalent reset state, a method for computing the new initial state when possible is reported in [50]. It uses implicit state enumeration techniques (see Section 3.2.2) to find a cycle of states containing the reset state. Simulation along the cycle is used to determine the new initial state for the retimed circuit.

Recently the notion of a resetable circuit was proposed [37]. Resetable circuits have an easily computed initializing sequence which brings the machine to a known state independent of the state on power-up. Although retiming preserves the resetable property, Boolean operations may not. At a later date the initial state re-computation after retiming may be combined in SIS with initializing sequences, so that an optimal choice can be made.

Retiming Don't Cares

Don't care conditions, used for minimization of the logic function at each node in a network, are computed based on other nodes in the network. For sequential circuits, this is done by considering each combinational logic block between register boundaries as a separate network: the inputs and outputs of each block consist of primary inputs and outputs as well as register outputs and inputs. The don't care conditions are computed based on the boundaries of these blocks.

During retiming, the registers are moved and these boundaries change. This invalidates the computed don't care information, even though the combinational logic function at each node is unchanged. An algorithm for preserving the **maximal** don't care sets during retiming has been reported [46] and is currently being implemented in SIS (it is not available in the current release of SIS). Although often the don't care information can be re-computed based on the new structure of the logic blocks, in some cases this results in a smaller don't care set. Some of the don't cares that depend on the original structure must be translated to don't cares for the retimed structure or they are lost. In particular, external, or user-specified don't cares cannot be re-computed based on the new logic blocks and should be preserved during retiming.

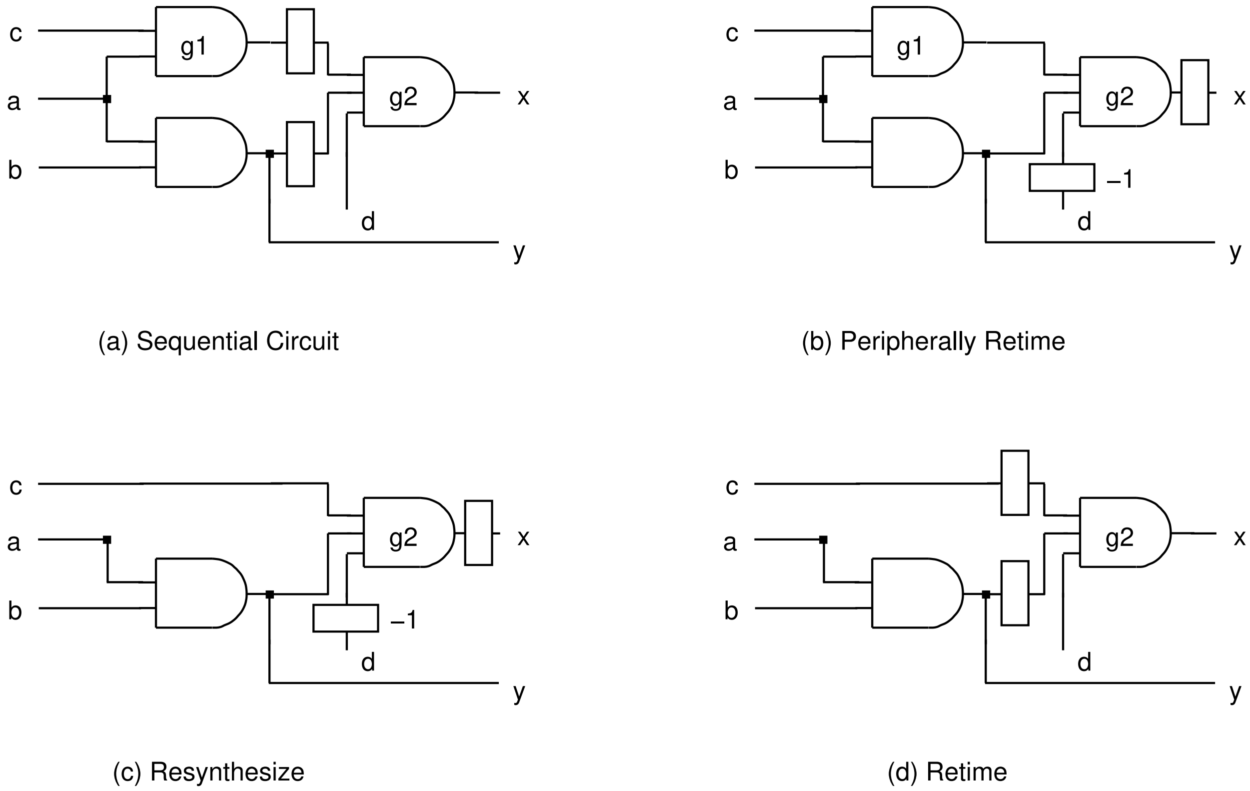


Figure 8: Retiming and Resynthesis Example

As a by-product of the investigation on retiming don't cares, an algorithm was developed for propagating don't care conditions **forward** in a network. That is, given the don't cares for the inputs of a node, an appropriate don't care function is computed for the output of the node. Techniques for propagating don't cares backward have already been developed in the work on ODC's [42]. Together, the forward and backward propagation algorithms can be used to propagate don't care conditions completely throughout a network, regardless of where these don't cares originate.

Retiming and Resynthesis

Retiming finds optimal register positions without altering the combinational logic functions at each node. Standard combinational techniques optimize a logic network, but given a sequential circuit, the optimization is limited to each combinational block: interactions between signals across register boundaries are not exploited.

These two techniques are combined in the retiming and resynthesis algorithm [28]. The first step is to identify the largest subcircuits that can be peripherally retimed, i.e. that can be retimed in such a way as to move all the registers to the boundaries of the subcircuit. Peripheral retiming is an extended notion of retiming where "negative" latches at the boundary are allowed (time is borrowed from the environment) as long as they can be retimed away after resynthesis. After peripheral retiming, standard combinational techniques are used to minimize the interior logic. Finally, the registers are retimed back into the circuit to minimize cycle time or the number of registers used. An example of this procedure is shown in Figure 8. In the example, pure combinational optimization would not yield any improvement, and standard retiming as it was first described cannot be applied to the circuit. It is only by combining peripheral retiming, which allows negative registers, and resynthesis that the improvement is obtained.

This algorithm has been successful in performance optimization of pipelined circuits [29]. Experiments with area minimization are ongoing, but it is expected that good results will be obtained only when the latest combinational optimization techniques, which use a larger set of observability don't cares, are employed. In particular, those that use a maximal observability don't care set can exploit the signal dependencies exposed during peripheral retiming. Retiming and resynthesis will be implemented in SIS after further investigation and experimentation.

Technology Mapping

The strategy of representing feedback lines by latches preserves the acyclic nature of the circuit representation and allows the combinational technology mapping algorithms to be easily extended to sequential mapping [33]. The same tree-covering algorithm [39] is used but the pattern matching procedure is extended to accommodate sequential elements. The pattern matching relies no longer on just the network topology but also on the type of library element and on the location of latch input and output pins. By choosing to treat the problem of selecting a clocking scheme (single-phase synchronous, multi-phase synchronous or even asynchronous) and clocking parameters as a higher level issue, the output of sequential technology mapping can be controlled by simply restricting the types of sequential elements allowed in the technology library. This approach can handle complex latches which are either synchronous or asynchronous, but not both (for example, synchronous latches with asynchronous set/reset) [3].

Implicit State Enumeration Techniques

Recently, methods for implicitly enumerating the states in a finite-state machine have been reported and can be used to explore the state space in very large examples [10, 53]. Such techniques employ a breadth-first traversal of sets of states of a machine. Beginning with an initial state set, all the states reachable in one transition from that set are computed simultaneously, then all the states reachable in two transitions, etc. Efficient implementation of this technique requires careful computation of the product of the transition relation (present-state/next-state relation) and the current set of states. This can be done either by reducing the size of the resulting function during computation by eliminating variables appropriately [53], or by using a divide and conquer approach [10]. Each function is efficiently implemented and manipulated using BDDs [6].

These techniques are useful for generating sequential don't care conditions. The reachable state set is computed with implicit techniques; the complement of this set represents the unreachable states. These are don't care conditions expressed in terms of the registers in the circuit. In SIS the command **extract_seq_dc** extracts the unreachable states and stores them as external don't cares. Subsequent calls to **full_simplify** (see Section 3.2.1) exploit these don't cares.

Implicit state enumeration techniques are also useful for verification. The equivalence of two finite-state machines (logic-level implementations) can be verified by computing the reachable states for the product machine. The two machines may have different state encodings. At each iteration, the states reached are checked for equivalent output functions. The **verify_fsm** command in SIS has verified machines with more than 10^{68} states [53].

3.3 Signal Transition Graph Manipulations

The design input to SIS need not be a synchronous specification. An asynchronous specification can be given as a signal transition graph. In this section, algorithms are outlined for the synthesis of hazard-free circuits under different delay models from signal transition graphs.

If a signal transition graph is *correct* (live, safe, free choice and with a consistent labeling), then it can be automatically *synthesized* if it satisfies another condition:

- *complete state coding*: whenever two markings have the same label, then they must enable the same set of *output transitions*.

The synthesis is performed by exhaustive simulation of all reachable markings. This produces a *state graph*, where each marking is associated with a state, and each edge, representing a transition firing, joins the corresponding predecessor and successor markings. For example, in Figure 3(c) the state graph fragment corresponding to Figure 3(a) is shown, where the state labeled 0100 corresponds to p_1 being marked.

If the signal transition graph is correct and has complete state coding, then we can derive from the state graph an *implied value* for each output signal in each state as follows. For every state s, for every output signal x, the implied value of x in s is:

[3] Such latches cannot be represented by a combinational network feeding a latch element; the require some additional logic at the latch output. This introduces cyclic dependency to the cost function, and the tree-covering can no longer be solved by a dynamic programming algorithm.

- the value of x in the label of s if no transition for x is enabled in the marking corresponding to s. For example, in state 0100, corresponding to p_1 being marked, no transition for c is enabled, so the implied value of c is 0.
- otherwise, the complement of that value. For example, in state 0100 transition b_2^- is enabled, so the implied value of b in 0100 is 0.

A combinational function, computing the implied value for each input label (minterm) obviously implements the required behavior of each output signal. This straightforward implementation, though, can still exhibit *hazards* (i.e. temporary deviations from the expected values, due to the non-zero delays of gates and wires in a combinational circuit). SIS provides an implementation of the algorithms described in [32] and [20] to produce hazard-free circuits under the *unbounded gate-delay* model and the *bounded wire-delay* model respectively.

The commands that operate on a signal transition graph and produce an asynchronous circuit implementation of it can be roughly classified as *analysis* and *synthesis* commands.

Analysis commands

- **astg_current** prints some information on the current signal transition graph.
- **astg_marking** prints the current marking, if any, of the signal transition graph (it is useful if the marking was obtained automatically by SIS , since this is an expensive computation).
- **astg_print_sg** prints the state graph obtained during the synthesis process (mainly useful for debugging purposes).
- **astg_print_stat** prints a brief summary of the signal transition graph and state graph characteristics.

Synthesis commands

- **astg_syn** synthesizes a hazard-free circuit using the *unbounded gate-delay* model. It eliminates hazards from the initial circuit, produced from the implied values as described above, by adding redundant gates and/or adding input connections to some gates, as described in [32]. This command (as well as **astg_to_f**) can compute the initial marking if it is not explicitly given.
- **astg_to_f** synthesizes a circuit (like **astg_syn**), but in addition it analyzes and stores its *potential hazards* using the *bounded wire-delay* model[4]. Those hazards can be eliminated, after a *constrained logic synthesis* and *technology mapping*, by the **astg_slow** command.
- **astg_slow** inserts delays, as described in [20], in order to produce a hazard-free circuit in a specific implementation technology. Note that this step can take advantage of the knowledge about *minimum delays in the environment*[5]. These delays can either be given as a single lower bound, with the **-d** option, or in a file, with the **-f** option. Each line in this environment delay file contains a pair of signals and a pair of numbers, and represents the minimum guaranteed lower bound on the delay between a transition on the first signal and a rising/falling transition respectively on the second signal.

  In order to do worst-case hazard analysis, **astg_slow** also provides a rudimentary mechanism to describe gates with upper and lower bounds on the delay, using the **-m** option. This option specifies a multiplicative factor to obtain the *minimum* delay of each gate, given its maximum delay as specified by the gate library used for technology mapping.

  Note that the **astg_to_f** and **astg_slow** commands require that only *algebraic* operations during logic synthesis in order to guarantee hazard-free synthesis. Namely only the **sweep**, **fx**, **gkx**, **gcx**, **resub**, **eliminate**, **decomp** and **map** commands can be used. Other commands such as **simplify** and **full_simplify** may re-introduce hazards.

[4]Note that a circuit that does not have hazards with the unbounded gate delay model, where all wire delays are assumed to be zero, can exhibit hazards with the bounded wire-delay model. The designer should use the delay model that best suits the underlying implementation technology (see [7]).

[5]By default the environment is assumed to respond instantaneously, since this represents a pessimistic worst-case assumption for hazard analysis.

TLU Synthesis	
xl_ao	cube-packing on an infeasible network
xl_k_decomp	apply Roth-Karp decomposition
xl_split	modified kernel extraction
xl_imp	apply different decomposition schemes and pick the best
xl_absorb	reduce infeasibility of the network
xl_cover	use binate covering to minimize number of blocks
xl_partition	minimize number of blocks by collapsing nodes into immediate fanouts
xl_part_coll	initial mapping and partial collapse on the network
xl_merge	identify function-pairs to be placed on the same CLB
xl_decomp_two	cube-packing targeted for two-output CLB's
xl_rl	reduce number of levels
_xl_nodevalue	print infeasible nodes
MB Synthesis	
act_map	block count minimization for *act1* and *act* architectures

Table 1: PGA Synthesis Commands in SIS

- **astg_contract**, given a specific output signal x for which logic needs to be synthesized, removes from the signal transition graph all the signals that are not directly required as inputs of the logic implementing x. Note that contraction is guaranteed to yield a contracted signal transition graph *with complete state coding* only if the original signal transition graph was a *marked graph*.

- **astg_lockgraph**, given a correct signal transition graph without complete state coding, uses the algorithm described in [56] to produce a signal transition graph with less concurrency but complete state coding (so that it can be synthesized). Note that this technique works successfully only for *marked graphs*.

- **astg_persist** ensures that the signal transition graph is *persistent*, i.e. no signal that enables a transition can change value before that transition has fired (this property ensures a somewhat simpler implementation, at the cost of a loss in concurrency). Note that this command works successfully only for *marked graphs*.

- **astg_to_stg** produces a *state transition graph* representation that describes a behavior equivalent to the signal transition graph. It can be used, for example, to compare a synchronous and an asynchronous implementation of (roughly) the same specification.

3.4 Logic Synthesis for Programmable Gate Arrays (PGA's)

Synthesizing circuits targeted for particular PGA architectures requires new architecture-specific algorithms. A number of these have been implemented in latter versions of MISII, and are being improved and enhanced in SIS. These algorithms are described briefly in this section. A summary of the commands in SIS for PGA synthesis are given in Table 1.

The basic PGA architectures share a common feature: repeated arrays of identical logic blocks. A *logic block* (or *basic block*) is a versatile configuration of logic elements that can be programmed by the user. There are two main categories of block structures, namely *Table-Look-Up (TLU)* and *Multiplexor-Based (MB)*; the resulting architectures are called the TLU and the MB architectures respectively. A basic block of a TLU architecture (also called configurable logic block or CLB) implements any function having up to m inputs, $m \geq 2$. For a given TLU architecture, m is a fixed number. A typical example is the Xilinx architecture [18], in which $m = 5$. In MB architectures, the basic block is a configuration of multiplexors [13]. In SIS, combinational circuits can be synthesized for both these architectures. Area minimization for both architectures and delay minimization for TLU architectures are supported. We are working on extending the capabilities to handle sequential circuits.

The algorithms and the corresponding commands that implement them in SIS are described next. For complete details of the algorithms, the reader may refer to [34, 35, 36].

3.4.1 Synthesis for TLU architectures

If a function has at most m inputs, we know it can be realized with one block. We call such a function m-*feasible*. Otherwise, it is m-infeasible. A network is m-feasible if the function at each node of the network is m-feasible. Let us first address the problem of minimizing the number of basic blocks used to implement a network. We assume that the technology-independent optimizations have been already applied on the given network description.

In order to realize an infeasible network using the basic blocks, we have to first make infeasible nodes feasible. After obtaining a feasible network, we have to minimize the number of blocks used. These two subproblems are described next.

Making an infeasible node feasible

Decomposition is a way of obtaining smaller nodes from a large node. We use decomposition to obtain a set of m-feasible nodes for an infeasible node function. The following is a summary of various decomposition commands for TLU architectures that are in SIS:

xl_ao	cube-packing on an infeasible network
xl_k_decomp	apply Roth-Karp decomposition
xl_split	modified kernel extraction
xl_imp	apply different decomposition schemes and pick the best

Cube-packing [14] used in **xl_ao** treats the cubes (product-terms) of the function as items. The size of each cube is the number of literals in it. Each logic block is a bin with capacity m. The decomposition problem can then be seen as that of packing the cubes into a minimum number of bins.

xl_k_decomp uses classical Roth-Karp decomposition [38] to decompose an infeasible node into m-feasible nodes. A set X (called the *bound set*) of cardinality m is chosen from the inputs of the infeasible function f. The rest of the inputs form the *free set* Y. Then the Roth-Karp technique decomposes f as follows:

$$f(X,Y) = g(\alpha_1(X), \alpha_2(X), \ldots, \alpha_p(X), Y) \tag{1}$$

where $\alpha_1, \alpha_2, \ldots, \alpha_p$ are m-feasible functions. If $p + |Y| > m$, g is infeasible and needs to be decomposed further. To get the best possible decomposition, all possible choices of bound sets have to be tried. With options `-e` and `-f`, all bound sets of a node are tried and the one resulting in the minimum number of nodes is selected.

xl_split extracts kernels from an infeasible node. This procedure is recursively applied on the kernel and the node, until either they become m-feasible or no more kernels can be extracted, in which case a simple and-or decomposition is performed.

xl_imp tries to get the best possible decomposition for the network. It applies a set of decomposition techniques on each infeasible node n of the network. These techniques include cube-packing on the sum-of-products form of n, cube-packing on the factored form of n, Roth-Karp decomposition, and kernel extraction. The best decomposition result - the one which has a minimum number of m-feasible nodes - is selected.

One command which does not necessarily generate an m-feasible network, but reduces the *infeasibility* of a network is **xl_absorb**. Infeasibility of a network is measured as the sum of the number of fanins of the infeasible nodes. The command **xl_absorb** moves the fanins of the infeasible nodes to feasible nodes so as to decrease the infeasibility of the network. Roth-Karp decomposition is used to determine if a fanin of a node could be made to fan in to another node.

Block Count Minimization

After decomposition, an m-feasible network is obtained and may be mapped directly onto the basic blocks. However, it may be possible to collapse some nodes into their fanouts while retaining feasibility. The following commands in SIS do the block count minimization:

xl_cover	use binate covering
xl_partition	collapse nodes into immediate fanouts

xl_cover enumerates all sets of nodes which can be realized as a single block. Each such set is called a *supernode*. The enumeration is done using the maximum flow algorithm repeatedly. A smallest subset S of supernodes that satisfy the following three constraints is selected:

1. each node of the network should be included in at least one supernode in S,
2. each input to a supernode in S should either be an output of another supernode in S, or an external input to the network, and
3. each output of the network should be an output of some supernode in S.

This is a binate covering formulation [39]. Mathony's algorithm [30] is used to solve this formulation. For large networks, this algorithm is computationally intensive and several heuristics for fast approximate solutions are used (option `-h`).

xl_partition tries to reduce the number of nodes by collapsing nodes into their immediate fanouts. It also takes into account extra nets created. It collapses a node into its fanout only if the resulting fanout is m-feasible. It associates a cost with each (node, fanout) pair which reflects the extra nets created if node is collapsed into the fanout. It then selects pairs with lowest costs and collapses them. With `-t` option, a node is considered for collapsing into all the fanouts, and is collapsed if all the fanouts remain m-feasible after collapsing. The node is then deleted from the network. Further optimization results by considering the technique of moving of fanins (using `-m` option). This technique is applied as follows. Before considering the collapse of node n into its fanout(s), we check if any fanin F of n could be moved to G - another fanin of n. This increases the chances of n being collapsed into its fanout(s). Moreover, it may later enable some other fanin of n to be collapsed into n.

Overall Algorithm

We first apply technology-independent optimization on the given network. Each infeasible node of this network is mapped as follows. We first decompose it into an m-feasible subnetwork using the techniques discussed above. The block count minimization is then applied on this subnetwork. Applying decomposition and block count minimization node-by-node gives better results than first applying decomposition on the entire network and then using block count minimization. The reason is that the exact block count minimization on the full network is computationally intensive, and hence is run only in heuristic mode.

The node-by-node mapping paradigm does not exploit structural relationship *between* the nodes of the network. This is achieved in command **xl_part_coll** (for *partial collapse*) by collapsing each node into its fanouts, remapping the fanouts, and computing the gain from this collapse. A greedy approach is employed - the collapse is accepted if it results in a gain.

Next we apply a heuristic block count minimization on the entire network (for example **xl_cover -h 3**).

For good results, the following script may be used:

```
xl_part_coll -m -g 2
xl_coll_ck
xl_partition -m
simplify
xl_imp
xl_partition -t
xl_cover -e 30 -u 200
xl_coll_ck -k
```

For very fast results, the following script may be used:

```
xl_ao
xl_partition -tm
```

xl_coll_ck collapses a feasible network if the number of primary inputs is small (specified by `-c` option), applies Roth-Karp decomposition and cofactoring schemes, picks the best result and compares with the original network (before collapsing). If the number of nodes is smaller, accepts the better decomposition. It does nothing if m = 2. If -k option is specified, Roth-Karp decomposition is not applied; only cofactoring is used.

Note that in all the commands, the default value of m is 5. It may be changed by specifying -n option to each command. Intermediate points in the quality-time trade-off curve may be obtained by suitably modifying the scripts.

One useful command not described above is **_xl_nodevalue -v support**. It prints nodes that have at least **support** fanins. This command is used to make sure that a feasible network has indeed been obtained. For example, if $m = 5$, use **_xl_nodevalue -v 6**.

Handling Two Outputs

There are special features in some TLU architectures. For example, in the Xilinx architecture, a CLB may implement one function of at most 5 inputs, or two functions if both of them have at most 4 inputs each and the total number of inputs is at most 5 (in which case they are called *mergeable*). The problem is to minimize the number of two-output CLB's used for a given circuit description. SIS has two special commands for handling two-output CLB's:

xl_merge	identify function-pairs to be placed on the same block
xl_decomp_two	cube-packing targeted for two-output CLB's

The approach currently followed in SIS is to minimize first the number of single-output blocks using the script(s) described above and then use **xl_merge** as a post-processing step to place maximum number of mergeable function-pairs in one CLB each. This reduces to the following problem: *"Given a feasible network, find the largest set of disjoint pairs of mergeable functions."* We have shown that this problem can be formulated as the maximum cardinality matching problem in a related graph. An exact solution can be generated using *lindo*, an integer linear programming package. If *lindo* is not found in the path, **xl_merge** switches to a heuristic to solve the matching problem.

A different approach that sometimes gives better results is the following. First obtain a 4-feasible network (by running any one of the above scripts with -n 4) and then use **xl_merge** without -F option. Since there are no nodes with 5 inputs, all nodes can potentially pair with other nodes. As a result the number of matches is higher. When -F option is not used, **xl_merge** first finds maximum number of mergeable function pairs and then applies block minimization on the subnetwork consisting of unmatched nodes of the network. This sometimes results in further savings. We recommend that the user should use scripts for both 4-feasible and 5-feasible, apply **xl_merge** and pick the network that uses lower number of CLB's.

The command **xl_decomp_two** does decomposition of the network targeted for two-output CLB's. It is a modification of the cube-packing approach. However, it does not guarantee a feasible network; other decomposition commands should be run afterwards to ensure feasibility.

Timing Optimization

xl_rl performs delay optimizations during logic synthesis. Given a feasible network (preferably generated by **speed_up**), **xl_rl** reduces the number of levels of TLU blocks used in the network. Then any block count minimization command (e.g. **xl_cover, xl_partition**) can be applied to reduce the number of blocks without increasing the number of levels. The details of the complete algorithm may be found in [36].

3.4.2 Synthesis for MB architectures

We have implemented synthesis algorithms for Actel's *act1* architecture (Figure 9). The command is called **act_map**. The architecture *act* (that is, *act1* with the OR gate removed) is also supported. No library needs to be read.

The outline of the algorithm [34] is as follows: first, for each node of the network a BDD (ordered or unordered) is constructed. The basic block of the architecture is represented with pattern graphs. The problem of covering the BDD with minimum number of pattern graphs is solved by first decomposing the BDD into trees and then mapping each tree onto the pattern-graphs using a dynamic programming algorithm. The overall algorithm is very similar to the one for TLU architecture. After initial mapping, an iterative improvement phase of partial collapse, decomposition and quick phase is entered. Quick phase decides if it is better to implement a node in the complement form. The user may specify number of iterations, limits on number of fanins of nodes for collapsing or decomposition. A bdnet-like file may be generated in which case the mapped network consists of nodes that are implemented by one basic block each.

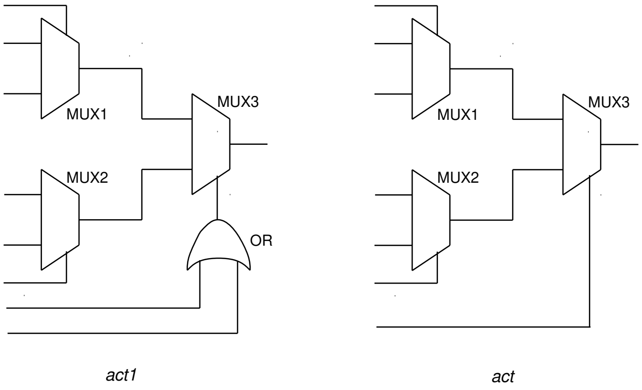


Figure 9: Two multiplexor-based architectures

3.5 Summary of SIS

Some of the new commands in SIS and their functions are summarized in Table 2. Currently SIS is being exercised on a number of standard and industrial examples, and being tested at various academic and industrial locations. The algorithms, particularly simplification and verification, are being run through large examples to test their robustness in terms of CPU time and memory usage.

New techniques are being explored for overcoming current limitations in the size of the circuits that can be manipulated. One direction for improvement is BDD ordering. Since the ordering of variables for constructing a BDD has a direct effect on the size of the BDD, with improved algorithms for determining the variable ordering, the size of the BDDs will decrease allowing larger circuits to be synthesized with these techniques.

The sequential technology mapping algorithm supports both synchronous and asynchronous latches, but not synchronous latches with asynchronous signals (e.g. set, reset). These latches cannot be represented by a combinational network feeding a latch element; some additional logic is required at the latch output. This introduces a cyclic dependency to the cost function, and the tree-covering can no longer be solved by a dynamic programming algorithm. Solutions to this problem are under investigation. One possibility is to tag every primary input and output node as either synchronous or asynchronous. Then, a latch with asynchronous set/reset signals can be represented as a latch with only synchronous set/reset signals, and the mapper distinguishes between the two by the types of input pins.

A project is underway to store an un-encoded state machine in an MDD representation (multiple-valued decision diagram) [49] rather than the cumbersome STG representation; state assignment algorithms that work from multi-level logic descriptions represented as MDDs are being explored. Also under investigation are techniques for sequential test pattern generation. While the BLIF format allows a hierarchical net-list to be given, currently the hierarchy is flattened to a single sequential circuit in SIS. This limitation is being addressed in the development of a hierarchical version of SIS.

4 Part II: Using the SIS Design Environment

4.1 Synchronous Synthesis Example

The synthesis of a sequential circuit using SIS is best illustrated with an example. The example chosen is **mark1**, which is in the MCNC benchmark set [26]. The specification is given in the KISS format (part of a smaller example in KISS is embedded in the BLIF specification of Figure 2).

```
sis> read_kiss mark1.kiss2

mark1.kiss2     pi= 5   po=16   nodes=  0       latches= 0
lits(sop)=   0  lits(fac)=   0  #states(STG)=  16
```

<u>STG Manipulation</u>	
one_hot	quick state assignment using one_hot encoding
state_assign	find an optimal binary code for each state and create a corresponding network
state_minimize	minimize the number of states by merging compatible states
stg_cover	verify that the STG behavior contains the logic behavior
stg_extract	extract an STG from a logic implementation
<u>Combinational Optimization</u>	
full_simplify	simplify the two-level function at each node using external, satisfiability, and observability don't cares
fx	extract common single-cube and double-cube divisors
red_removal	remove combinational redundancies
reduce_depth	improve performance by selectively reducing the circuit depth
<u>Sequential Optimization</u>	
randr	optimize using combined retiming and resynthesis
retime	move the registers for minimum cycle or minimum number of registers
<u>ASTG Analysis</u>	
astg_current	print ASTG information
astg_marking	print initial marking
astg_print_sg	print state graph (after synthesis)
astg_print_stat	print other ASTG information
<u>ASTG Synthesis</u>	
astg_syn	synthesize hazard-free implementation with unbounded gate-delay
astg_to_f	synthesize and do hazard analysis with bounded wire-delay
astg_slow	remove hazards (after mapping) with bounded wire-delay
astg_contract	reduce the ASTG to the signals required for a single output
astg_lockgraph	ensure complete state coding (marked graph only)
astg_persist	ensure persistency (marked graph only)
astg_to_stg	produce state transition graph equivalent to ASTG
<u>Miscellaneous</u>	
atpg	test pattern generation on combinational logic (assumes a scan design)
extract_seq_dc	extract unreachable states and store as external don't cares
verify_fsm	verify the equivalence of two state machines using implicit state enumeration techniques

Table 2: Partial List of New SIS commands

Some statistics are given: the name of the circuit, the number of primary inputs and outputs, the number of nodes in the multi-level logic implementation, the number of latches, the number of literals in sum-of-product and factored form, and the number of states in the STG. The number of literals is the number of variables in true and complemented form that appear in the Boolean functions for all nodes. The number of literals in factored form is roughly equal to the number of transistor pairs that would be required for a static CMOS implementation. Since only the STG data structure is present, the only statistics reported are number of inputs and outputs and the number of states in the STG.

State minimization is performed on the circuit with the STAMINA program

```
sis> state_minimize stamina

Running stamina, written by June Rho, University of Colorado at Boulder
Number of states in original machine : 16
Number of states in minimized machine : 12

mark1.kiss2     pi= 5   po=16   nodes=  0       latches= 0
lits(sop)=   0  lits(fac)=   0  #states(STG)=  12
```

and the number of states in the STG is reduced from 16 to 12.

State assignment determines binary codes for each state and substitutes these codes for the symbolic states, creating a logic-level implementation.

```
sis> state_assign jedi

mark1.kiss2     pi= 5   po=16   nodes= 20       latches= 4
lits(sop)= 195  lits(fac)= 144  #states(STG)=  12
```

The JEDI state assignment program targets a multi-level implementation, The number of latches reflects the number of encoding bits used by the state assignment program. There are four unused state codes; these will later be extracted and used as don't care conditions during the optimization.

Next the standard script from MISII is called, which iteratively extracts common cubes and factors, resubstitutes them in the node functions, and collapses nodes whose literal-count savings is below a threshold. The result is a circuit with a significant improvement in the number of literals.

```
sis> source script

mark1.kiss2     pi= 5   po=16   nodes= 16       latches= 4
lits(sop)=  91  lits(fac)=  81  #states(STG)=  12
```

No further improvement can be obtained by subsequent invocations of the standard MISII script. The powerful node simplification procedure is called to minimize the Boolean functions at each node using a large don't care set, which includes both ODC's and SDC's.

```
sis> full_simplify

mark1.kiss2     pi= 5   po=16   nodes= 16       latches= 4
lits(sop)=  91  lits(fac)=  80  #states(STG)=  12
```

In this case, the improvement is insignificant, but further improvements can be obtained by running the simplification algorithm with sequential don't cares. These don't cares are unreachable states computed using implicit state enumeration techniques.

```
sis> extract_seq_dc

number of latches = 4     depth = 6     states visited = 12
```

During this process, 12 states are visited in breadth-first order, and the remaining 4 are stored as don't care conditions. These are exploited by **full_simplify**.

```
sis> full_simplify

mark1.kiss2     pi= 5   po=16   nodes= 16       latches= 4
lits(sop)=  79  lits(fac)=  70  #states(STG)=  12
```

The series of commands executed above demonstrate the improvements that the **full_simplify** algorithm with sequential don't cares obtains over the **simplify** algorithm in the MISII standard script. However, our recommendation is to use a new script, **script.rugged** instead of the MISII standard script. The new script extracts factors faster using **fx**, and hence will quickly reduce the size of large examples. Initially it uses the faster but less powerful MISII minimization; **full_simplify** is used later in the script for powerful node minimization. For the **mark1** example, a better result than the one shown above can be obtained automatically by invoking **script.rugged**. The following commands are applied to **mark1** after state assignment.

```
sis> extract_seq_dc
sis> source script.rugged

mark1.kiss2     pi= 5   po=16   nodes= 17       latches= 4
lits(sop)=  70  lits(fac)=  66  #states(STG)=  12
```

The retiming algorithm can be run to improve the cycle time of the circuit.

```
sis> retime -n
Lower bound on the cycle time =  3.40
Retiming will minimize the cycle time
RETIME: Initial clk = 13.60, Desired clk = 3.40
initial cycle delay         = 13.60
initial number of registers = 4
initial logic cost          = 78.00
initial register cost       = 4.00

Failed at 3.40  : Now attempting 8.50
Failed at 8.50  : Now attempting 11.05
Failed at 11.05 : Now attempting 12.32
Failed at 12.32 : Now attempting 12.96
Failed at 12.96 : Now attempting 13.28
Failed at 13.28 : Now attempting 13.44
Success at 13.44, Delay is 13.40
Success: Now attempting 13.34
Quitting binary search at 13.34

final cycle delay         = 13.40
final number of registers = 8
final logic cost          = 78.00
final register cost       = 8.00

RETIME: Final cycle time  achieved = 13.40
mark1.kiss2     pi= 5   po= 16   nodes= 17       latches= 8
lits(sop)=  70  lits(fac)=  66
```

The result is that the cycle time decreases from 13.6 to 13.4, while the number of latches increases from 4 to 8. In this example, the increase in latches outweighs the small decrease in cycle time. However, in many cases only a small increase in the number of latches is required for a significant improvement in cycle time. Moreover, the user can control the tradeoff between latch count and cycle time.

Finally, a library of gates including latches is read in.

```
sis> rlib lib2.genlib
```

lib2.genlib is based on a standard-cell design style and is from the MCNC benchmark set [26]. It has been augmented with latches. The circuit is mapped into a net-list of gates and latches in this target technology.

```
sis> map -s
total gate area:        104864.00
maximum arrival time: (19.75,20.35)

mark1.kiss2     pi= 5   po=16   nodes= 50       latches= 9
lits(sop)= 120  lits(fac)= 105
```

The number of literals changes as the mapping routine restructures the network to fit the target technology. The gate area is based on standard-cell grid count, and the arrival time is computed with static timing analysis, using the delays on the gates given in the library. The two numbers represent the maximum rise and fall delays.

A new script, **script.delay**, is provided with SIS and targets performance optimization. It should be used after area optimization is complete. If it is invoked for **mark1** after the rugged script is used, the result is as follows:

```
total gate area:        116000.00
maximum arrival time: ( 8.08, 8.19)
```

This is compared with the best results for area optimization, which, in the example above, is the circuit obtained after the last application of **script.rugged**:

```
total gate area:         79344.00
maximum arrival time: (15.82,16.33)
```

The use of **script.delay** resulted in a 50% improvement in speed for a 46% penalty in area. Again, the user can control the trade-off between area and delay by varying the parameters supplied to the various algorithms.

4.2 Asynchronous Synthesis Example

Let us examine a simple example of a signal transition graph from [8], shown in Figure 10. It is a marked graph (no place has multiple predecessors or successors), and no initial marking is given.

```
.model two_phase_fifo
.inputs Ri Ao D
.outputs Ro Ai L
.graph
D- L-/1 Ro-
L-/1 Ai-
Ai- Ri+
Ri+ L+/1
Ro- Ao-
Ao- L+/1
L+/1 D+
D+ L-/2 Ro+
L-/2 Ai+
Ai+ Ri-
Ri- L+/2
```

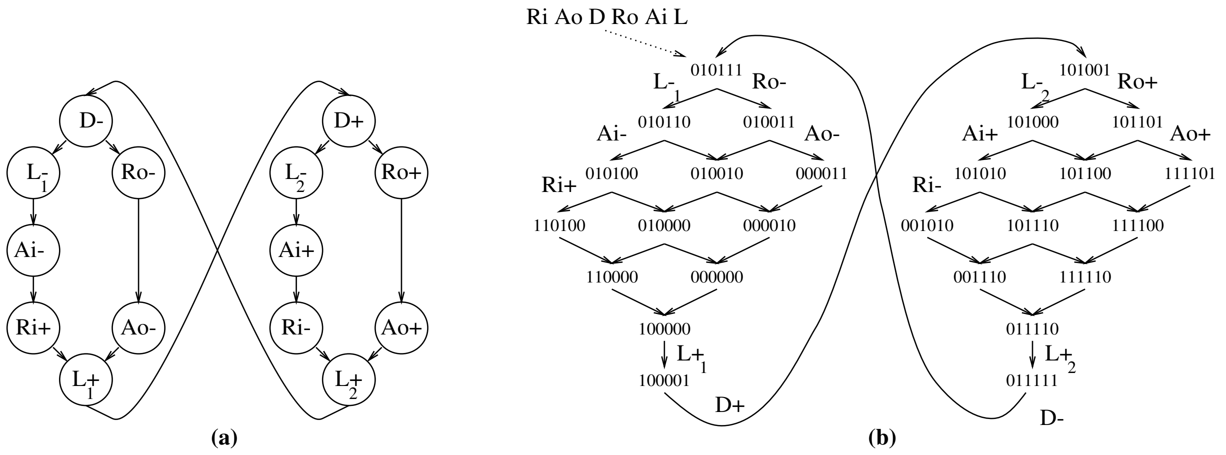


Figure 10: A Signal Transition Graph and its State Graph

```
Ro+ Ao+
Ao+ L+/2
L+/2 D-
.end
```

Suppose that the signal transition graph is contained in a file called `chu150.g`. The following SIS session shows an example of how to use the main commands that use the *unbounded gate-delay* model.

```
sis> read_astg chu150.g
sis> astg_syn
sis> print
     Ai_next = Ai_ D + Ai_ {L} + D {L}'
     {L} = Ao D Ri' + Ao' D' Ri
     {Ro} = D
     {Ai} = Ai_
sis> print_latch
input: {Ai_next} output: Ai_ init val: 1 cur val: 1 type: as control: none
sis> print_io
primary inputs:   Ri Ao D Ai_
primary outputs:  {Ai_next} {Ro} {Ai} {L}
sis> astg_current
two_phase_fifo
        File: chu150.g (modified)
        Pure: Y  Place-simple: Y
        Connected: Y  Strongly Connected: Y
        Free Choice: Y  Marked Graph: Y  State Machine: N
        SM components: 4
        MG components: 1
sis> astg_marking
{<L+/2,D->}
sis> astg_print_stat
File Name = chu150.g
Total Number of Signals = 6 (I = 3/O = 3)
Initial State = state 55 : (Ri=0 Ao=1 D=1 Ro=1 Ai=1 L=1 )
        enabled transitions : [D- ]
```

```
Total Number of States = 26
sis>
```

astg_syn automatically computes an initial marking, printed with **astg_marking**, that corresponds to the (implicit) place between L_2^+ and D^- being marked, and hence to the state labeled 011111 in the state graph.

Note that the function of signal *Ai* depends on the signal itself. In order to break the asynchronous feedback loop, an asynchronous latch is inserted between primary output *Ai_next* and primary input *Ai_*. So each output signal that is part of an asynchronous feedback loop is represented inside SIS by three distinct signals:

- a "fictitious" primary input (distinguished by *"_"*),
- a "fictitious" primary output (distinguished by *"_next"*),
- a "real" primary output.

All three signals, though, are actually directly connected to each other, since an asynchronous latch represents only a *delay element* used to break the loop. This representation trick is required to describe an asynchronous circuit within a framework better suited for synchronous systems.

Signals *L* and *Ro*, on the other hand, are purely combinational, hence they are just represented as a combinational logic block.

Now we can examine a similar session using the *bounded wire-delay* model commands.

```
sis> read_astg chu150.g
sis> astg_to_f
sis> print
     {Ro_next} = D
     [15] = Ai_ D + Ai_ L_ + D L_'
     {Ai} = Ai_
     [17] = Ao D Ri' + Ao' D' Ri
     {L} = L_
sis> print_latch
input: {[15]} output: Ai_ init val: 1 cur val: 3 type: as control: none
input: {[17]} output: L_ init val: 1 cur val: 3 type: as control: none
sis> print_io
primary inputs:   Ri Ao D Ai_ L_
primary outputs:  {Ro_next} {Ai} {L} {[15]} {[17]}
sis> read_library asynch.genlib
sis> map -W
sis> print_latch
input: {[587]} output: Ai_ init val: 1 cur val: 3 type: as control: none
input: {[586]} output: L_ init val: 1 cur val: 3 type: as control: none
sis> print_io
primary inputs:   Ri Ao D Ai_ L_
primary outputs:  {Ro_next} {Ai} {L} {[587]} {[586]}
sis> print_gate -s
ao33:combinational :    1 (area=64.00)
delay:asynch    :    1 (area=0.00)
inv:combinational :    3 (area=16.00)
nor2:combinational :    2 (area=24.00)
sr_nor:asynch   :    1 (area=40.00)
Total: 8 gates, 200.00 area
sis> print_delay -ap 1 o()
 ... using library delay model
```

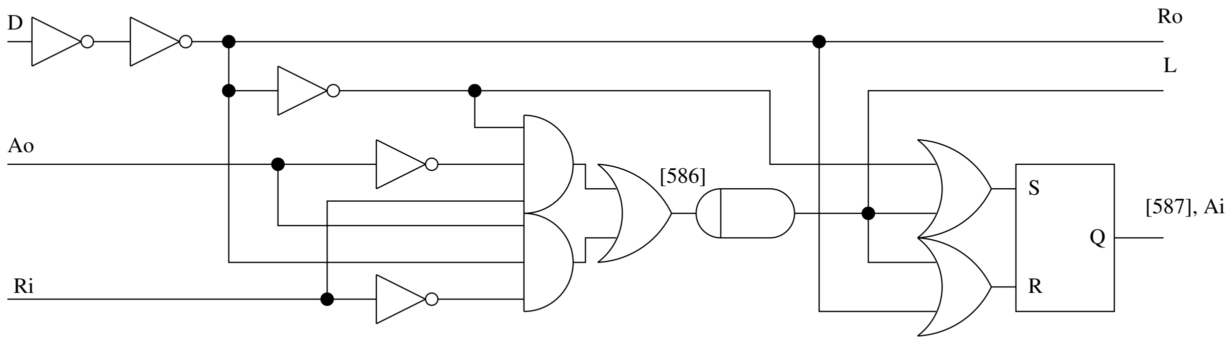


Figure 11: An implementation of Figure 10

```
{[587]}   : arrival=( 4.60  4.60)
sis> astg_slow -d .8
Slowing down D (fails by 0.60)
sis> print_gate -s
ao33:combinational :    1 (area=64.00)
delay:asynch    :    1 (area=0.00)
inv:combinational :    5 (area=16.00)
nor2:combinational :    2 (area=24.00)
sr_nor:asynch   :    1 (area=40.00)
Total: 10 gates, 232.00 area
sis> print_delay -ap 1 o()
 ... using library delay model
{[587]}   : arrival=( 7.00  7.00)
sis>
```

The synthesized circuit is shown in Figure 11, where the delay element symbol denotes an asynchronous latch that is implemented by a simple wire. The delays reported by **print_delay** are measured within the combinational logic only, so the longest path of 7.0 units at the end is between nodes *D* and *[586]*.

Note that in the bounded wire-delay case asynchronous latches are used to implement every signal (not just self-dependent ones), in order to be able to trace delays across the network. The only exception is signal *Ro*, whose function is just a wire. The circuit after technology mapping has a hazard due to the early arrival of signal *D* if we use a delay model where the minimum delay across a gate is 80% of its maximum delay (as specified by **-d .8**). So **astg_slow** inserts a pair of inverters after *D*, thus increasing both the area and the delay of the circuit in order to eliminate all hazards.

If on the other hand, using our knowledge of the environment behavior, we could assume that the minimum reaction time of the environment is 2 time units (as specified by **-t 2**), then the circuit would be hazard-free without the need to insert any delay:

```
sis> read_astg chu150.g
sis> astg_to_f
sis> read_library asynch.genlib
sis> map -W
sis> astg_slow -d .8 -t 2
sis> print_gate -s
ao33:combinational :    1 (area=64.00)
delay:asynch    :    1 (area=0.00)
inv:combinational :    3 (area=16.00)
nor2:combinational :    2 (area=24.00)
```

```
sr_nor:asynch   :    1 (area=40.00)
Total: 10 gates, 200.00 area
sis> print_delay -ap 1 o()
 ... using library delay model
{[587]}   : arrival=( 4.60  4.60)
sis>
```

5 Conclusions

SIS is a comprehensive system for synthesizing and optimizing sequential circuits. It is easily interfaced with other design tools through widely used interchange formats, and integrates a number of state-of-the-art algorithms for synthesis, optimization, testing, and verification. Several scripts are provided to automate the synthesis process and to serve as a starting point for experimentation with the parameters of each algorithm. SIS targets specifically a sequential circuit design methodology and thereby can exploit properties specific to sequential circuits.

6 Acknowledgements

SIS is a joint project supported by the professors (in particular Professors Brayton, Newton, and Sangiovanni-Vincentelli) and many of the graduate students in the CAD group at U.C. Berkeley. Many of the ideas about sequential and combinational logic synthesis have originated in the advanced classes and research seminars within the group. Many students have contributed code and participated in the development of SIS : Luciano Lavagno, Sharad Malik, Cho Moon, Rajeev Murgai, Alex Saldanha, Hamid Savoj, Ellen Sentovich, Narendra Shenoy, Tom Shiple, Kanwar Jit Singh, Moshe Steiner, Paul Stephan, Colin Stevens, Hervé Touati and Huey-Yih Wang. Several programs are distributed with SIS : JEDI (state assignment) contributed by Bill Lin, NOVA (state assignment) contributed by Tiziano Villa, and STAMINA (state minimization) contributed by June-Kyung Rho at the University of Colorado at Boulder. Tony Ma wrote the SENUM program upon which the STG extraction algorithm is based. We gratefully acknowledge the support of NSF under grant EMC-8419744, DARPA under grant JFBI90-073, the SRC, the state of California MICRO program, Intel, AT&T, DEC, and Motorola.

References

[1] *VAX DECSIM Reference Manual*. Digital Equipment Corporation, December 1985. Not generally available.

[2] Karen A. Bartlett, Robert K. Brayton, Gary D. Hachtel, Reily M. Jacoby, Christopher R. Morrison, Richard L. Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. Multilevel Logic Minimization Using Implicit Don't Cares. *IEEE Transactions on Computer-Aided Design*, 7(6):723–740, June 1988.

[3] R.K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, May 1982.

[4] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[5] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[6] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35, August 1986.

[7] J. A. Brzozowski and C-J. Seger. Advances in Asynchronous Circuit Theory – Part I: Gate and Unbounded Inertial Delay Models. *Bulletin of the European Association of Theoretical Computer Science*, October 1990.

[8] T. A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[9] T. A. Chu. Synthesis of Hazard-free Control Circuits from Asynchronous Finite State Machine Specifications. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 1–10, 1992.

[10] O. Coudert, C. Berthet, and J.C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[11] G. DeMicheli. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):63–73, January 1991.

[12] Andrea Casotto editor. Octtools-5.1 Manuals. In *UCB Electronics Research Lab*, September 1991.

[13] A. Gamal et. al. An Architecture for Electrically Configurable Gate Arrays. *IEEE Journal of Solid-State Circuits*, 24(2):394–398, April 1989.

[14] R.J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs. In *Proceedings of the Design Automation Conference*, pages 227–233, June 1991.

[15] G. De Micheli. Synchronous logic synthesis. In *International Workshop on Logic Synthesis*, page 5.2, North Carolina, May 1989.

[16] Abhijit Ghosh. *Techniques for Test Generation and Verification in VLSI Sequential Circuits*. UCB PhD Thesis, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, December 1991.

[17] G.D. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. In *The Proceedings of the European Conference on Design Automation*, pages 184–191, Amsterdam, The Netherlands, February 1991.

[18] Xilinx Inc. Xilinx Programmable Gate Array User's Guide, 1988.

[19] T. Larrabee. Efficient Generation of Test Patterns Using Boolean Difference. In *Proceedings of the International Test Conference*, pages 795–801, 1989.

[20] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for Synthesis of Hazard-free Asynchronous Circuits. In *Proceedings of the Design Automation Conference*, pages 302–308, June 1991.

[21] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Advanced Research in VLSI: Proceedings of the Third Caltech Conference*, pages 86–116. Computer Science Press, 1983.

[22] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. In *TM 372, MIT/LCS, 545 Technology Square, Cambridge, Massachusetts 02139*, October 1988.

[23] C.E. Leiserson and J.B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[24] B. Lin and A.R. Newton. Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages. In *Proceedings of the International Conference on VLSI*, pages 187–196, August 1989.

[25] B. Lin, H. Touati, and A.R. Newton. Don't Care Minimization of Multi-level Sequential Logic Networks. In *Proceedings of the International Conference on Computer-Aided Design*, pages 414–417, November 1990.

[26] R. Lisanke. Logic synthesis benchmark circuits for the International Workshop on Logic Synthesis, May, 1989.

[27] A. Malik, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. A Modified Approach to Two-level Logic Minimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 106–109, November 1988.

[28] S. Malik, E.M. Sentovich, R.K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimization of sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design*, 10(1):74–84, January 1991.

[29] S. Malik, K.J. Singh, R.K. Brayton, and A. Sangiovanni-Vincentelli. Performance Optimization of Pipelined Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 410–413, November 1990.

[30] H.-J. Mathony. Universal Logic Design Algorithm and its Application to the Synthesis of Two-level Switching Circuits. *IEE Proceedings*, 136 Pt. E(3), May 1989.

[31] M.C. Paull and S.H. Unger. Minimizing the Number of States in Incompletely Specified Sequential Switching Functions. *IRE Transactions on Electronic Computers*, EC-8:356–367, September 1959.

[32] C. W. Moon, P. R. Stephan, and R. K. Brayton. Synthesis of Hazard-free Asynchronous Circuits from Graphical Specifications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 322–325, November 1991.

[33] C.W. Moon, B. Lin, H. Savoj, and R.K. Brayton. Technology Mapping for Sequential Logic Synthesis. In *Proc. Int'l. Workshop on Logic Synthesis*, North Carolina, May 1989.

[34] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni Vincentelli. Logic Synthesis for Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 620–625, June 1990.

[35] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni Vincentelli. Improved Logic Synthesis Algorithms for Table Look Up Architectures. In *Proceedings of the International Conference on Computer-Aided Design*, pages 564–567, November 1991.

[36] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni Vincentelli. Performance Directed Synthesis for Table Look Up Programmable Gate Arrays. In *Proceedings of the International Conference on Computer-Aided Design*, pages 572–575, November 1991.

[37] C. Pixley and G. Beihl. Calculating Resetability and Reset Sequences. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[38] J.P. Roth and R.M. Karp. Minimization over Boolean Graphs. In *IBM Journal of Research and Development*, April 1982.

[39] R. Rudell. *Logic Synthesis for VLSI Design*. Memorandum No. UCB/ERL M89/49, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, April 1989.

[40] Alexander Saldanha, Albert Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Multi-Level Logic Simplification using Don't Cares and Filters. In *Proceedings of the Design Automation Conference*, pages 277–282, 1989.

[41] H. Savoj and R.K. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *Proceedings of the Design Automation Conference*, pages 297–301, June 1990.

[42] H. Savoj, R.K. Brayton, and H. Touati. Extracting Local Don't Cares for Network Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[43] H. Savoj, H.-Y. Wang, and R.K. Brayton. Improved Scripts in MIS-II for Logic Minimizaton of Combinational Circuits. In *The International Workshop on Logic Synthesis*, May 1991.

[44] James B. Saxe. *Decomposable Searching Problems and Circuit Optimization by Retiming: Two Studies in General Transformations of Computational Structures*. CMU-CS-85-162, Carnegie-Mellon University, Department of Computer Science, August 1985.

[45] R. Segal. *BDSYN: Logic Description Translator; BDSIM: Switch-Level Simulator*. Master's Thesis, Memorandum No. UCB/ERL M87/33, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1987.

[46] E.M. Sentovich and R.K. Brayton. Preserving Don't Care Conditions During Retiming. In *Proceedings of the International Conference on VLSI*, pages 461–470, August 1991.

[47] K.J. Singh and A. Sangiovanni-Vincentelli. A Heuristic Algorithm for the Fanout Problem. In *Proceedings of the Design Automation Conference*, pages 357–360, June 1990.

[48] K.J. Singh, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 282–285, November 1988.

[49] A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton. Algorithms for Discrete Function Manipulation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[50] H. Touati and R.K. Brayton. Computing the Initial States of Retimed Circuits. *IEEE Transactions on Computer-Aided Design*, July 1992. To appear.

[51] H. Touati, C. Moon, R.K. Brayton, and A.Wang. Performance-Oriented Technology Mapping. In *Proceedings of the sixth MIT VLSI Conference*, pages 79–97, April 1990.

[52] H. Touati, H. Savoj, and R.K. Brayton. Delay Optimization of Combinational Logic Circuits by Clustering and Partial Collapsing. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[53] H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[54] Hervé J. Touati. *Performance-Oriented Technology Mapping*. Memorandum No. UCB/ERL M90/109, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, November 1990.

[55] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley Interscience, 1969.

[56] P. Vanbekbergen, G. Goossens, and H. De Man. A Local Optimization Technique for Asynchronous Control Circuits. In *Proc. Int'l. Workshop on Logic Synthesis*, North Carolina, May 1991.

[57] J. Vasudevamurthy and J. Rajski. A Method for Concurrent Decomposition and Factorization of Boolean Expressions. In *Proceedings of the International Conference on Computer-Aided Design*, pages 510–513, November 1990.

[58] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations. *IEEE Transactions on Computer-Aided Design*, 9(9):905–924, September 1990.

Appendix A : Berkeley Logic Interchange Format (BLIF)

The goal of BLIF is to describe a logic-level hierarchical circuit in textual form. A circuit is an arbitrary combinational or sequential network of logic functions. A circuit can be viewed as a directed graph of combinational logic nodes and sequential logic elements. Each node has a two-level, single-output logic function associated with it. Each feedback loop must contain at least one latch. Each net (or signal) has only a single driver, and either the signal or the gate which drives the signal can be named without ambiguity.

In the following, angle-brackets surround nonterminals, and square-brackets surround optional constructs.

1 Models

A model is a flattened hierarchical circuit. A BLIF file can contain many models and references to models described in other BLIF files. A model is declared as follows:

```
.model <decl-model-name>
.inputs <decl-input-list>
.outputs <decl-output-list>
.clock <decl-clock-list>
<command>
     .
     .
     .
<command>
.end
```

decl-model-name is a string giving the name of the model.

decl-input-list is a white-space-separated list of strings (terminated by the end of the line) giving the formal input terminals for the model being declared. If this is the first or only model, then these signals can be identified as the primary inputs of the circuit. Multiple *.inputs* lines are allowed, and the lists of inputs are concatenated.

decl-output-list is a white-space-separated list of strings (terminated by the end of the line) giving the formal output terminals for the model being declared. If this is the first or only model, then these signals can be identified as the primary outputs of the circuit. Multiple *.outputs* lines are allowed, and the lists of outputs are concatenated.

decl-clock-list is a white-space-separated list of strings (terminated by the end of the line) giving the clocks for the model being declared. Multiple *.clock* lines are allowed, and the lists of clocks are concatenated.

command is one of:

```
<logic-gate>         <generic-latch>       <library-gate>
<model-reference>    <subfile-reference>   <fsm-description>
<clock-constraint>   <delay-constraint>
```

Each *command* is described in the following sections.

The BLIF parser allows the *.model, .inputs, .outputs, .clock* and *.end* statements to be optional. If *.model* is not specified, the *decl-model-name* is assigned the name of the BLIF file being read. It is an error to use the same string for *decl-model-name* in more than one model. If *.inputs* is not specified, it can be inferred from the signals which are not the outputs of any other logic block. Similarly, *.outputs* can be inferred from the signals which are not the inputs to any other blocks. If any *.inputs* or *.outputs* are given, no inference is made; a node that is not an output and does not fanout produces a warning message.

If *.clock* is not specified (e.g., for purely combinational circuits) there are no clocks. *.end* is implied at end of file or upon encountering another *.model*.

Important: the first model encountered in the main BLIF file is the one returned to the user. The only *.clock, clock-constraint,* and *timing-constraint* constructs retained are the ones in the first model. All subsequent models can be incorporated into the first model using the *model-reference* construct.

Anywhere in the file a '#' (hash) begins a comment that extends to the end of the current line. Note that the character '#' cannot be used in any signal names. A '\' (backslash) as the last character of a non-comment line indicates concatenation of the subsequent line to the current line. No whitespace should follow the '\'.

Example:

```
.model simple
.inputs a b
.outputs c
.names a b c        #    .names described later
11 1
.end
```

```
#   unnamed model
.names a b  \
c                   #    '\' here only to demonstrate its use
11 1
```

Both models "simple" and the unnamed model describe the same circuit.

2 Logic Gates

A *logic-gate* associates a logic function with a signal in the model, which can be used as an input to other logic functions. A *logic-gate* is declared as follows:

```
.names <in-1> <in-2> ... <in-n> <output>
<single-output-cover>
```

output is a string giving the name of the gate being defined.

in-1, in-2, ... in-n are strings giving the names of the inputs to the logic gate being defined.

single-output-cover is, formally, an n-input, 1-output PLA description of the logic function corresponding to the logic gate. $\{0, 1, -\}$ is used in the n-bit wide "input plane" and $\{0, 1\}$ is used in the 1-bit wide "output plane". The ON-set is specified with 1's in the "output plane," and the OFF-set is specified with 0's in the "output plane." The DC-set is specified for primary output nodes only, by using the construct *.exdc*.

A sample *logic-gate* with its *single-output-cover*:

```
.names v3 v6 j u78 v13.15
1--0 1
-1-1 1
0-11 1
```

In a given row of the *single-output-cover*, "1" means the input is used in uncomplemented form, "0" means the input is complemented, and "–" means not used. Elements of a row are ANDed together, and then all rows are ORed.

As a result, if the last column (the "output plane") of the *single-output-cover* is all 1's, the first n columns (the "input plane") of the *single-output-cover* can be viewed as the truth table for the logic gate named by the string *output*. The order of the inputs in the *single-output-cover* is the same as the order of the strings *in-1, in-2, ..., in-n* in the *.names* line. A space between the columns of the "input plane" and the "output plane" is required.

The translation of the above sample *logic-gate* into a sum-of-products notation would be as follows:

```
v13.15 = (v3 u78') + (v6 u78) + (v3' j u78)
```

To assign the constant "0" to some logic gate `j`, use the following construct:

```
.names j
```

To assign the constant "1", use the following:

```
.names j
1
```

The string *output* can be used as the input to another *logic-gate* before the *logic-gate* for *output* is itself defined.

For a more complete description of the PLA input format, see *espresso(5)*.

3 External Don't Cares

External don't cares are specified as a separate network within a model, and are specified at the end of the model specification. Each external don't care function, which is specified by a *.names* construct, must be associated with a primary output of the main model and specified as a function of the primary inputs of the main model (hierarchical specification of external don't cares is currently not supported).

The external don't cares are specified as follows:

```
.exdc
.names <in-1> <in-2> ... <in-n> <output>
<single-output-cover>
```

exdc indicates that the following *.names* constructs apply to the external don't care network.

output is a string giving the name of the primary output for which the conditions are don't cares.

in-1, in-2, ... in-n are strings giving the names of the primary inputs which the don't care conditions are expressed in terms of.

single-output-cover is an n-input, 1-output PLA description of the logic function corresponding to the don't care conditions for the output.

The following is an example circuit with external don't cares:

```
.model a
.inputs x y
.outputs j
.subckt b x=x y=y j=j
.exdc
.names x j
1 1
.end

.model b
.inputs x y
.outputs j
.names x y j
11 1
.end
```

The translation of the above example into a sum-of-products notation would be as follows:

```
j = x * y;
external d.c. for j = x;
```

4 Flip flops and latches

A *generic-latch* is used to create a delay element in a model. It represents one bit of memory or state information. The *generic-latch* construct can be used to create any type of latch or flip-flop (see also the *library-gate* section). A *generic-latch* is declared as follows:

```
.latch <input> <output> [<type> <control>] [<init-val>]
```

input is the data input to the latch.

output is the output of the latch.

type is one of {fe, re, ah, al, as}, which correspond to "falling edge," "rising edge," "active high," "active low," or "asynchronous."

control is the clocking signal for the latch. It can be a *.clock* of the model, the output of any function in the model, or the word "NIL" for no clock.

init-val is the initial state of the latch, which can be one of $\{0, 1, 2, 3\}$. "2" stands for "don't care" and "3" is "unknown." Unspecified, it is assumed "3."

If a latch does not have a controlling clock specified, it is assumed that it is actually controlled by a single global clock. The behavior of this global clock may be interpreted differently by the various algorithms that may manipulate the model after the model has been read in. Therefore, the user should be aware of these varying interpretations if latches are specified with no controlling clocks.

Important: All feedback loops in a model must go through a *generic-latch*. Purely combinational-logic cycles are not allowed.

Examples:

```
.inputs d              # a clocked flip-flop
.output q
.clock c
.latch d q re c 0
.end

.inputs in             # a very simple sequential circuit
.outputs out
.latch out in 0
.names in out
0 1
.end
```

5 Library Gates

A *library-gate* creates an instance of a technology-dependent logic gate and associates it with a node that represents the output of the logic gate. The logic function of the gate and its known technology dependent delays, drives, etc. are stored with the *library-gate*. A *library-gate* is one of the following:

```
.gate <name> <formal-actual-list>
.mlatch <name> <formal-actual-list> <control> [<init-val>]
```

name is the name of the *.gate* or *.mlatch* to instantiate. A gate or latch with this name must be present in the current working library.

formal-actual-list is a mapping between the formal parameters of *name* (the terminals of the *library-gate*) and the actual parameters of the current model (any signals in this model). The format for a *formal-actual-list* is a white-space-separated sequence of assignment statements of the form:

```
formal1=actual1 formal2=actual2 ...
```

All of the formal parameters of *name* must be specified in the *formal-actual-list* and the single output of *name* must be the last one in the list.

control is the clocking signal for the mlatch, which can be either a *.clock* of the model, the output of any function in the model, or the word "NIL" for no clock.

init-val is the initial state of the mlatch, which can be one of $\{0, 1, 2, 3\}$. "2" stands for "don't care" and "3" is "unknown." Unspecified, it is assumed "3."

A *.gate* refers to a two-level representation of an arbitrary input, single output gate in a library. A *.gate* appears under a technology-independent interpretation as if it were a single *logic-gate*.

A *.mlatch* refers to a latch (not necessarily a D flip flop) in a library. A *.mlatch* appears under a technology-independent interpretation as if it were a single *generic-latch* and possibly a single *logic-gate* feeding the data input of that *generic-latch*.

*.gate*s and *.mlatch*es are used to describe circuits that have been implemented using a specific library of standard logic functions and their technology-dependent properties. The library of *library-gate*s must be read in before a BLIF file containing *.gate* or *.mlatch* constructs is read in.

The string *name* refers to a particular gate or latch in the library. The names "nand2," "inv," and "jk_rising_edge" in the following examples are descriptive names for gates in the library. The following BLIF description:

```
.inputs v1 v2
.outputs j
.gate nand2 A=v1 B=v2 O=x  # given: formals of this gate are A, B, O
.gate inv A=x O=j          # given: formals of this gate are A & O
.end
```

could also be specified in a technology-independent way (assuming "nand2" is a 2-input NAND gate and "inv" is an INVERTER) as follows:

```
.inputs v1 v2
.outputs j
.names v1 v2 x
0- 1
-0 1
.names x j
0 1
.end
```

Similarly:

```
.inputs j kbar
.outputs out
.clock clk
.mlatch jk_rising_edge J=j K=k Q=q clk 1   # given: formals are J, K, Q
.names q out
0 1
.names kbar k
0 1
.end
```

could have been specified in a technology-independent way (assuming "jk_rising_edge" is a JK rising-edge-triggered flip flop) as follows:

```
.inputs j kbar
.outputs out
.clock clk
.latch temp q re clk 1     # the .latch
.names j k q temp          # the .names feeding the D input of the .latch
-01 1
1-0 1
.names q out
0 1
.names kbar k
0 1
.end
```

6 Model (subcircuit) references

A *model-reference* is used to insert the logic functions of one model into the body of another. It is defined as follows:

```
.subckt <model-name> <formal-actual-list>
```

model-name is a string giving the name of the model being inserted. It need not be previously defined in this file, but should be defined somewhere in either this file, a *.search* file, or a master file that is *.search*ing this file. (see *.search* below)

formal-actual-list is a mapping between the formal terminals (the *decl-input-list, decl-output-list,* and *decl-clock-list*) of the called model *model-name* and the actual parameters of the current model. The actual parameters may be any signals in the current model. The format for a *formal-actual-list* is the same as its format in a *library-gate*.

A *.subckt* construct can be viewed as creating a copy of the logic functions of the called model *model-name*, including all of *model-name*'s *generic-latch*es, in the calling model. The hierarchical nature of the BLIF description of the model does not have to be preserved. Subcircuits can be nested, but cannot be self-referential or create a cyclic dependency.

Unlike a *library-gate*, a *model-reference* is not limited to one output.

The formals need not be specified in the same order as they are defined in the *decl-input-list, decl-output-list,* or *decl-clock-list*; elements of the lists can be intermingled in any order, provided the names are given correctly. Warning messages are printed if elements of the *decl-input-list* or *decl-clock-list* are not driven by an actual parameter or if elements of the *decl-output-list* do not fan out to an actual parameter. Elements of the *decl-clock-list* and *decl-input-list* may be driven by any logic function of the calling model.

Example: rather than rewriting the entire BLIF description for a commonly used subcircuit several times, the subcircuit can be described once and called as many times as necessary:

```
.model 4bitadder
.inputs A3 A2 A1 A0 B3 B2 B1 B0 CIN
.outputs COUT S3 S2 S1 S0
.subckt fulladder a=A0 b=B0 cin=CIN     s=S0 cout=CARRY1
.subckt fulladder a=A3 b=B3 cin=CARRY3  s=S3 cout=COUT
.subckt fulladder b=B1 a=A1 cin=CARRY1  s=XX cout=CARRY2
.subckt fulladder a=JJ b=B2 cin=CARRY2  s=S2 cout=CARRY3
                  # for the sake of example,
.names XX S1      # formal output 's' does not fanout to a primary output
1 1
.names A2 JJ      # formal input 'a' does not fanin from a primary input
1 1
.end

.model fulladder
.inputs a b cin
.outputs s cout
.names a b k
10 1
01 1
.names k cin s
10 1
01 1
.names a b cin cout
11- 1
1-1 1
-11 1
.end
```

7 Subfile References

A *subfile-reference* is:

```
.search <file-name>
```

file-name gives the name of the file to search.

A *subfile-reference* directs the BLIF reader to read in and define all the models in file *file-name*. A *subfile-reference* does not have to be inside of a *.model*. *subfile-reference*s can be nested.

Search files would usually be used to hold all the subcircuits referred to in *model-reference*s, while the master file merely searches all the subfiles and instantiates all the subcircuits it needs.

A *subfile-reference* is not equivalent to including the body of subfile *file-name* in the current file. It does not patch fragments of BLIF into the current file; it pauses reading the current file, reads *file-name* as an independent, self-contained file, then returns to reading the current file.

The first *.model* in the master file is always the one returned to the user, regardless of any *subfile-reference*s than may precede it.

8 Finite State Machine Descriptions

A sequential circuit can be specified in BLIF logic form, as a finite state machine, or both. An *fsm-description* is used to insert a finite state machine description of the current model. It is intended to represent the same sequential circuit as the current model (which contains logic), but in FSM form. The format of an *fsm-description* is:

```
.start_kiss
.i <num-inputs>
.o <num-outputs>
[.p <num-terms>]
[.s <num-states>]
[.r <reset-state>]
<input> <current-state> <next-state> <output>
              .
              .
              .
<input> <current-state> <next-state> <output>
.end_kiss
[.latch_order <latch-order-list>]
[<code-mapping>]
```

num-inputs is the number of inputs to the FSM, which should agree with the number of inputs in the *.inputs* construct for the current model.

num-outputs is the number of outputs of the FSM, which should agree with the number of outputs in the *.outputs* construct for the current model.

num-terms is the number of "<input> <current-state> <next-state> <output>" 4-tuples that follow in the FSM description.

num-states is the number of distinct states that appear in "<current-state>" and "<next-state>" columns.

reset-state is the symbolic name for the reset state for the FSM; it should appear somewhere in the "<current-state>" column.

input is a sequence of *num-inputs* members of $\{0, 1, -\}$.

output is a sequence of *num-outputs* members of $\{0, 1, -\}$.

current-state and *next-state* are symbolic names for the current state and next state transitions of the FSM.

latch-order-list is a white-space-separated sequence of latch outputs.

code-mapping is newline separated sequence of:

```
.code <symbolic-name> <encoded-name>
```

num-terms and *num-states* do not have to be specified. If the *reset-state* is not given, it is assigned to be the first state encountered in the "<current-state>" column.

The ordering of the bits in the *input* and *output* fields will be the same as the ordering of the variables in the *.inputs* and *.outputs* constructs if both an *fsm-description* and logic functions are given.

latch-order-list and *code-mapping* are meant to be used when both an *fsm-description* and a logical description of the model are given. The two constructs together provide a correspondence between the latches in the logical description and the state variables in the *fsm-description*. In a *code-mapping*, *symbolic-name* consists of a symbolic name from the "<current-state>" or "<next-state>" columns, and *encoded-name* is the pattern of bits ($\{0, 1\}$) that represent the state encoding for *symbolic-name*. The *code-mapping* should only be given if both an *fsm-description* and logic functions are given. *.latch-order* establishes a mapping between the bits of the *encoded-name*s of the *code-mapping* construct and the latches of the network. The order of the bits in the encoded names will be the same as the order of the latch outputs in the *latch-order-list*. There should be the same number of bits in the *encoded-name* as there are latches if both an *fsm-description* and a logical description are specified.

If both *logic-gate*s and an *fsm-description* of the model are given, the *logic-gate* description of the model should be consistent with the *fsm-description*, that is, they should describe the same circuit. If they are not consistent there will be no sensible way to interpret the model, which should then cause an error to be returned.

If only the *fsm-description* of the network is given, it may be run through a state assignment routine and given a logic implementation. A sole *fsm-description*, having no logic implementation, cannot be inserted into another model by a *model-reference*; the state assigned network, or a network containing both *logic-gate*s and an *fsm-description* can.

Example of an *fsm-description*:

```
.model 101        # outputs 1 whenever last 3 inputs were 1, 0, 1
.start_kiss
.i 1
.o 1
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.end
```

Above example with a consistent *fsm-description* and logical description:

```
.model
.inputs v0
.outputs v3.2
.latch    [6] v1   0
.latch    [7] v2   0
.start_kiss
.i 1
.o 1
.p 8
.s 4
.r st0
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.latch_order v1 v2
.code st0 00
.code st1 11
.code st2 01
.code st3 10
.names v0 [6]
1 1
.names v0 v1 v2 [7]
-1- 1
1-0 1
.names v0 v1 v2 v3.2
101 1
.end
```

9 Clock Constraints

A *clock-constraint* is used to set up the behavior of the simulated clocks, and to specify how clock events (rising or falling edges) occur relative to one another. A *clock-constraint* is one or more of the following:

```
.cycle <cycle-time>
.clock_event <event-percent> <event-1> [<event-2> ... <event-n>]
```

cycle-time is a floating point number giving the clock cycle time for the model. It is a unitless number that is to be interpreted by the user.

event-percent is a floating point number representing a percentage of the clock cycle time at which a specific *.clock_event* occurs. Fifty percent is written as “50.0.”

event-1 through *event-n* are one of the following:

```
<rise-fall>'<clock-name>
(<rise-fall>'<clock-name> <before> <after>)
```

where *rise-fall* is either “r” or “f” and stands for the rising or falling edge of the clock and *clock-name* is a clock from the *.clock* construct. The apostrophe between *rise-fall* and *clock-name* is a separator, and serves no purpose in and of itself.

before and *after* are floating point numbers in the same “units” as the *cycle-time* and are used to define the “skew” in the clock edges. *before* represents maximum amount of time before the nominal time that the edge can arrive; *after* represents the maximum amount of time after the nominal time that the edge can arrive. The nominal time is *event-percent*% of the *cycle-time*. In the unparenthesized form for the *clock-event*, *before* and *after* are assumed “0.0.”

All events, *event-1 ... event-n*, specified in a single *.clock_event* are to be linked together. A routine changing any one edge should also modify the occurrence time of all the related clock edges.
Example 1:

```
.clock clock1 clock2
.clock_event 50.0 r'clock1 (f'clock2 2.0 5.0)
```

Example 2:

```
.clock clock1 clock2
.clock_event 50.0 r'clock1
.clock_event 50.0 (f'clock2 2.0 5.0)
```

Both examples specify a nominal time of 50% of the cycle time, that the rising edge of clock1 must occur at exactly the nominal time, and that the falling edge of clock2 may occur from 2.0 units before to 5.0 units after the nominal time.

In Example 1, if r’clock1 is later moved to a different nominal time by some routine then f’clock2 should also be changed. However, in Example 2 changing r’clock1 would not affect f’clock2 even though they originally have the same value of *event-percent*.

10 Delay Constraints

A *delay-constraint* is used to specify parameters to more accurately compute the amount of time signals take to propagate from one point to another in a model. A *delay-constraint* is one or more of :

```
.area <area>
.delay <in-name> <phase> <load> <max-load> <brise> <drise> <bfall> <dfall>
.wire_load_slope <load>
.wire <wire-load-list>
.input_arrival <in-name> <rise> <fall> [<before-after> <event>]
.default_input_arrival <rise> <fall>
.output_required <out-name> <rise> <fall> [<before-after> <event>]
.default_output_required <rise> <fall>
.input_drive <in-name> <rise> <fall>
.default_input_drive <rise> <fall>
.output_load <out-name> <load>
.default_output_load <load>
```

rise, *fall*, *drive*, and *load* are all floating point numbers giving the rise time, fall time, input drive, and output load.

in-name is a primary input and *out-name* is a primary output.

before-after can be one of {b, a}, corresponding to "before" or "after," and *event* has the same format as the unparenthesized form of *event-1* in a *clock-constraint*.

.area sets the area of the model to be *area*.

.delay sets the delay for input *in-name*. *phase* is one of "INV," "NONINV," or "UNKNOWN" for inverting, non-inverting, or neither. *max-load* is a floating point number for the maximum load. *brise, drise, bfall,* and *dfall* are floating point numbers giving the block rise, drive rise, block fall, and drive fall for *in-name*.

.wire_load_slope sets the wire load slope for the model.

.wire sets the wire loads for the model from the list of floating point numbers in the *wire-load-list*.

.input_arrival sets the input arrival time for the input *in-name*. If the optional arguments are specified, then the input arrival time is relative to the *event*.

.output_required sets the output required time for the output *out-name*. If the optional arguments are specified, then the output required time is relative to the *event*.

.input_drive sets the input drive for the input *in-name*.

.output_load sets the output load for the output *out-name*.

.default_input_arrival, .default_output_required, .default_input_drive, .default_output_load set the corresponding default values for all the inputs/outputs whose values are not specifically set.

There is no actual unit for all the timing and load numbers. Special attention should be given when specifying and interpreting the values. The timing numbers are assumed to be in the same "unit" as the *cycle-time* in the *.cycle* construct.

Appendix B : Genlib Format

This is a description of the **genlib** format which is used to specify library gates in SIS.

1 Combinational Gate Specification

A cell is specified in the following format:

```
GATE <cell-name> <cell-area> <cell-logic-function>
<pin-info>
    .
    .
<pin-info>
```

<cell-name> is the name of the cell in the cell library. The resulting net-list will be in terms of these names.

<cell-area> defines the relative area cost of the cell. It is a floating point number, and may be in any unit system convenient for the user.

<cell-logic-function> is an equation written in conventional algebraic notation using the operators ’+’ for OR, ’*’ or nothing (space) for AND, ’!’ or ”’ (post-fixed) for NOT, and parentheses for grouping. The names of the literals in the equation define the input pin names for the cell; the name on the left hand side of the equation defines the output of the cell. The equation terminates with a semicolon.

Only single-output cells may be specified. The ’!’ operator may only be used on the input literals, or on the final output; it is not allowed internal to an expression. (This constraint may disappear in the future).

Also, the actual factored form is significant when a logic function has multiple factored forms. In principle, all factored forms could be derived for a given logic function automatically; this is not yet implemented, so each must be specified separately. Note that factored forms which differ by a permutation of the input variables (or by De Morgan’s law) are not considered unique.

Each *<pin-info>* has the format:

```
PIN <pin-name> <phase> <input-load> <max-load>
    <rise-block-delay> <rise-fanout-delay>
    <fall-block-delay> <fall-fanout-delay>
```

<pin-name> must be the name of a pin in the *<cell-logic-function>*, or it can be * to specify identical timing information for all pins.

<phase> is INV, NONINV, or UNKNOWN corresponding to whether the logic function is negative-unate, positive-unate, or binate in this input variable respectively. This is required for the separate rise-fall delay model. (In principle, this information is easily derived from the logic function; this field may disappear in the future).

<input-load> gives the input load of this pin. It is a floating point value, in arbitrary units convenient for the user.

<max-load> specifies a loading constraint for the cell. It is a floating point value specifying the maximum load allowed on the output.

<rise-block-delay> and *<rise-fanout-delay>* are the rise-time parameters for the timing model. They are floating point values, typically in the units nanoseconds, and nanoseconds/unit-load respectively.

<fall-block-delay> and *<fall-fanout-delay>* are the fall-time parameters for the timing model. They are floating point values, typically in the units nanoseconds, and nanoseconds/unit-load repsectively.

All of the delay information is specified on a pin-by-pin basis. The meaning is the delay information for the most critical pin is used to determine the delay for the gate.

2 Latch Specification

Latches are specified as follows:

```
LATCH <cell-name> <cell-area> <cell-logic-function>
<pin-info>
    .
    .
<pin-info>
<latch-spec>
[<clock-delay-info>]
[<constraint-info>]
```

<cell-name>, *<cell-area>*, *<cell-logic-function>* and *<pin-info>* are the same as in the combinational case.

<latch-spec> has the following format:

```
SEQ <latch-input> <latch-output> <latch-type>
```

<latch-input> must be the name of the output of the cell. Thus, *<latch-input>* is described as a function of *<latch-output>* in the *<cell-logic-function>*. A special name "ANY" can be used for *<latch-output>* to specify to the mapper that *<latch-output>* needs not be a signal name in the *<cell-logic-function>*. This is useful for describing D-type flip-flops and latches. *<latch-type>* can be ACTIVE_HIGH or ACTIVE_LOW for transparent latches; RISING_EDGE or FALLING_EDGE for edge-triggered flip-flops; or ASYNCH for asychronous latches.

<clock-delay-info> is necessary for synchronous latches and flip-flops. It gives the propagation delay from the clock pin to the output. Its format is as follows:

```
CONTROL <clock-pin-name> <input-load> <max-load>
    <rise-block-delay> <rise-fanout-delay>
    <fall-block-delay> <fall-fanout-delay>
```

<constraint-info> gives the setup and hold time constraints with respect to the clock signal. The format is:

```
CONSTRAINT <pin-name> <setup-time> <hold-time>
```

If not specified, the values of 0.0 are assumed.

3 Examples

Example 0 of latch specification:

```
LATCH "d-latch" 80 Q=D;
PIN D NONINV 1 999 1 .2 1 .2
SEQ Q ANY ACTIVE_HIGH
CONTROL CLK 1 999 1 .2 1 .2
CONSTRAINT D 0.2 0.2
```

Example 1 of <cell-logic-function>:

```
O = !(!I1 * I2);
```

This is legal and defines a NAND gate with 1 inverted input; it could also be specified as

```
O = I1 + !I2;
```

There is no advantage or disadvantage to either representation; only one need be given.

Example 2 of <cell-logic-function>:

```
O = !((I1 * I2) + !(I3 + I4));
```

is incorrectly specified. It must be re-written as either

```
O = !((I1 * I2) + (!I3 * !I4));
```

or

```
O = (!I1 + !I2)*(I3 + I4);
```

Again, there is no advantage or disadvantage to either representation as they differ only by De Morgan's law. Only one need be given. Note that there are no other factored forms for this equation.
Example 3 of <cell-logic-function>:

```
O = a*b + a*c + b*c;
```

This is one specification of a majority gate. Another possible specification (which IS considered different) is:

```
O = a*(b + c) + b*c;
```

Any permutation of the input variables does not provide a new specification; hence, these are the only two useful representations for a majority gate logic function. Both should be provided to the technology mapping program, until further notice.