Asynchronous Circuit Design. Chris J. Myers

ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

6

Muller Circuits

The infinite is the finite of every instant.

—Zen Saying

When I can't handle events, I let them handle themselves.

—Henry Ford

Life is pleasant. Death is peaceful. It's the transition that's troublesome.

—Isaac Asimov

We are ready for any unforeseen event that may or may not occur.

—Vice President Dan Quayle, 9/22/90

In this chapter we introduce the Muller school of thought to the synthesis of asynchronous circuits. *Muller circuits* are designed under the *unbounded gate delay model.* Under this model, circuits are guaranteed to work regardless of gate delays, assuming that wire delays are negligible. Muller circuit design requires explicit knowledge of the behaviors allowed by the environment. It does not, however, put any restriction on the speed of the environment.

The design of Muller circuits requires a somewhat different approach as compared with traditional sequential state machine design. Most synthesis methods for Muller circuits translate the higher-level specification into a state graph. Next, the state graph is examined to determine if a circuit can be generated using only the specified input and output signals. If two states are found that have the same values of inputs and outputs but lead through an output transition to different next states, no circuit can be produced di-

rectly. In this case, either the protocol must be changed or new internal state signals must be added to the design. The method of determining the needed state variables is quite different from that used for Huffman circuits. Next, logic is derived using modified versions of the logic minimization procedures described earlier. The modifications needed are based upon the technology that is being used for implementation. Finally, the design must be mapped to gates in a given gate library. This last step requires a substantially modified technology mapping procedure as compared with traditional state machine synthesis methods.

We first give a formal definition of speed independence. Then we describe the "state assignment" method for Muller circuits. Finally, we describe logic minimization and technology mapping, respectively.

6.1 FORMAL DEFINITION OF SPEED INDEPENDENCE

In order to design a speed-independent circuit, it is necessary to have complete information about the behavior of both the circuit being designed and the environment. Therefore, we restrict our attention to *complete circuits.* A complete circuit C is defined by a finite set of *states*, S. At any time, C is said to be in one of these states.

The behavior of a complete circuit is defined by the set of *allowed sequences* of states. Each allowed sequence can be either finite or infinite, and the set of allowed sequences can also be finite or infinite. For example, the sequence $(s_1, s_2, s_3, \ldots)$ says that state s_1 is followed by state s_2, but it does not state the time at which this state transition takes place. Therefore, in order to determine when a state transition takes place, it is necessary that consecutive states be different. In other words, for an allowed sequence $(s_1, s_2, \ldots)$, any pair of consecutive states $s_i \neq s_{i+1}$. Another property is that each state $s \in S$ is the initial state of at least one allowed sequence. One can also derive additional allowed sequences from known ones. For example, if $(s_1, s_2, s_3, \ldots)$ is an allowed sequence, then so is $(s_2, s_3, \ldots)$. If $(s_1, s_2, \ldots)$ and $(t_1, t_2, \ldots)$ are allowed sequences and $s_2 = t_1$, then $(s_1, t_1, t_2, \ldots)$ is also an allowed sequence.

Example 6.1.1 Consider a complete circuit composed of four states, $S = \{a, b, c, d\}$, which has the following two allowed sequences:

1. $a, b, a, b, \ldots$
2. a, c, d

The sequences above imply that the following sequences are also allowed:

1. $b, a, b, a, \ldots$
2. c, d
3. d

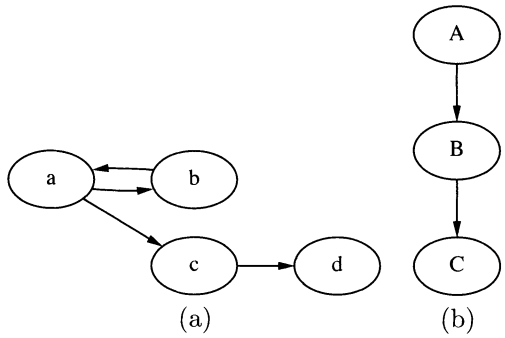

Fig. 6.1 (a) Simple state diagram. (b) Partial order of its equivalence classes.

4. a, b, a, c, d
5. a, b, a, b, a, c, d
6. b, a, c, d
7. etc.

A *state diagram* for this example is shown in Figure 6.1(a).

Two states $s_i, s_j \in S$ are *$\mathcal{R}$-related* (denoted $s_i \mathcal{R} s_j$) when:

1. $s_i = s_j$ or
2. s_i, s_j appear as a consecutive pair of states in some allowed sequence.

A sequence $(s_1, s_2, \ldots, s_m)$ is an *$\mathcal{R}$-sequence* if $s_i \mathcal{R} s_{i+1}$ for each $1 \leq i \leq m-1$.

A state s_i is *followed* by a state s_j (denoted $s_i \mathcal{F} s_j$) if there exists an $\mathcal{R}$-sequence $(s_i, \ldots, s_j)$. The $\mathcal{F}$-relation is reflexive and transitive, but not necessarily symmetric. If two states s_i and s_j are symmetric under the $\mathcal{F}$-relation (i.e., $s_i \mathcal{F} s_j$ and $s_j \mathcal{F} s_i$), they are said to be *equivalent* (denoted $s_i \mathcal{E} s_j$).

The equivalence relation, $\mathcal{E}$, partitions the finite set of states S of any circuit into equivalence classes of states. The $\mathcal{F}$-relation can be extended to these equivalence classes. If A and B are two equivalence classes, then $A \mathcal{F} B$ if there exists states $a \in A$ and $b \in B$ such that $a \mathcal{F} b$. Furthermore, if a is in the equivalence class A and b is in B and $A \mathcal{F} B$, then $a \mathcal{F} b$.

For any allowed sequence, there is a definite last class which is called the *terminal class.* A circuit C is *speed independent with respect to a state s* if all allowed sequences starting with s have the same terminal class.

Example 6.1.2 The circuit from Figure 6.1(a) is partitioned into three equivalence classes: $A = \{a, b\}$, $B = \{c\}$, and $C = \{d\}$. Applying the extended $\mathcal{F}$-relation to the equivalence classes, we get the following $A \mathcal{F} A$, $A \mathcal{F} B$, $A \mathcal{F} C$, $B \mathcal{F} B$, $B \mathcal{F} C$, and $C \mathcal{F} C$. This relation is depicted in Figure 6.1(b) without self-loops and transitive arcs. This circuit is not speed independent with respect to state a, since there exist allowed

sequences starting in a that end in terminal class A and others that end in terminal class C. However, this circuit is speed independent with respect to states c and d since all allowed sequences starting in these states end in terminal class C.

Muller circuits are typically modeled using a state graph (SG) derived from a higher-level graphical model such as a STG or TEL structure as described in Chapter 4. We can reformulate the notion of allowed sequences on a state graph. An allowed sequence of states $(s_1, s_2, \ldots)$ is any sequence of states satisfying the following three conditions:

1. No two consecutive states s_i and s_{i+1} are equal.
2. For any state s_{j+1} and signal u_i, one of the following is true:

$$\begin{aligned} s_{j+1}(i) &= s_j(i) \\ s_{j+1}(i) &= s'_j(i) \end{aligned}$$

3. If there exists a signal u_i and a state s_j such that $s_j(i) = s_r(i)$ and $s'_j(i) = s'_r(i)$ for all s_r in the sequence following s_j, then

$$s_j(i) = s'_j(i)$$

The second condition states that in an allowed sequence either a signal remains unchanged or changes to its implied value. The third condition states that if a signal is continuously excited to change, it eventually does change to its implied value.

Example 6.1.3 A simple speed-independent circuit and corresponding state graph is shown in Figure 6.2. Note that all states are contained in a single equivalence class, making it speed-independent.

6.1.1 Subclasses of Speed-Independent Circuits

There are several useful subclasses of speed-independent circuits. First, a circuit is *totally sequential with respect to a state s* if there is only one allowed sequence starting with s. Clearly, if there is only one allowed sequence starting with s, there can be only one terminal class which proves the following theorem.

Theorem 6.1 (Muller, 1959) *A circuit totally sequential with respect to s is also speed independent with respect to s.*

Example 6.1.4 A simple totally sequential circuit is shown in Figure 6.3(a), and its state graph is shown in Figure 6.3(b). In its initial state, $\langle 0R \rangle$, both x and y are 0, but y's implied value is 1. After y rises, the system moves to state $\langle R1 \rangle$, where x now has an implied value of 1. After x rises, the state is $\langle 1F \rangle$.

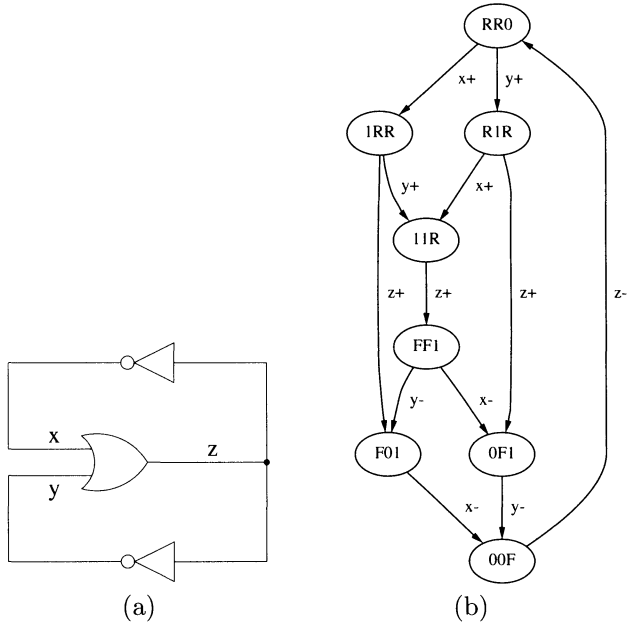

Fig. 6.2 (a) Speed-independent circuit. (b) Its state graph with state vector $\langle x, y, z \rangle$.

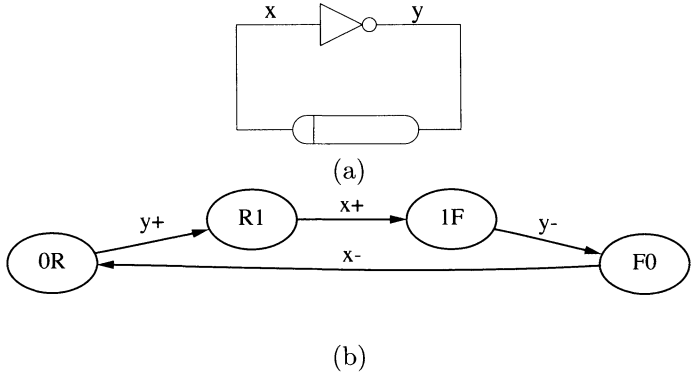

Fig. 6.3 (a) Totally sequential circuit. (b) Its state graph with state vector $\langle x, y \rangle$.

A circuit is *semi-modular* in a state s_i if in all states s_j reached after one signal has transitioned, any other signals excited in s_i are still excited in s_j. More formally:

$$\begin{aligned} & \forall t_1, t_2 \in T \;.\; (s_i, t_1, s_j) \in \delta \wedge (s_i, t_2, s_k) \in \delta \\ \Rightarrow \quad & \exists s_l \in S \;.\; (s_j, t_2, s_l) \in \delta \wedge (s_k, t_1, s_l) \in \delta \end{aligned}$$

A circuit is *semi-modular with respect to a state* s if all states reachable from s are semi-modular. A totally sequential circuit is semi-modular, but the converse is not necessarily true. A circuit that is semi-modular with respect to a state s is also speed independent with respect to s, but again the converse is not necessarily true. These results are summarized in the following theorems.

Theorem 6.2 (Muller, 1959) *A circuit totally sequential with respect to s is also semi-modular with respect to s.*

Theorem 6.3 (Muller, 1959) *A circuit semi-modular with respect to s is also speed independent with respect to s.*

> **Example 6.1.5** The circuit shown in Figure 6.2 is speed-independent, but it is not semi-modular. For example, in state $\langle 1RR \rangle$ signals y and z are both excited to rise, but after z rises, it goes to state $\langle F01 \rangle$, in which y is no longer excited. Another simple circuit and its corresponding state graph are shown in Figure 6.4. This circuit is semi-modular with respect to each state in the state graph.

Input transitions typically are allowed to be disabled by other input transitions, but output transitions are typically not allowed to be disabled. Therefore, another useful class of circuits are those which are *output semi-modular.* A SG is output semi-modular in a state s_i if only input signal transitions can disable other input signal transitions. More formally:

$$\begin{aligned} & \forall t_1 \in T_O \;.\; \forall t_2 \in T \;.\; (s_i, t_1, s_j) \in \delta \wedge (s_i, t_2, s_k) \in \delta \\ \Rightarrow \quad & \exists s_l \in S \;.\; (s_j, t_2, s_l) \in \delta \wedge (s_k, t_1, s_l) \in \delta \end{aligned}$$

where T_O is the set of output transitions (i.e., $T_O = \{u+, u- \mid u \in O\}$).

> **Example 6.1.6** If and only if all signals in the circuit shown in Figure 6.2 are inputs, then it is output semi-modular.

6.1.2 Some Useful Definitions

It is often useful to be able to determine in which states a signal is excited to rise or fall. The sets of *excitation states*, $ES(u+)$ and $ES(u-)$, provide this information and are defined as follows:

$$\begin{aligned} ES(u+) &= \{s \in S \mid s(u) = 0 \wedge u \in X(s)\} \\ ES(u-) &= \{s \in S \mid s(u) = 1 \wedge u \in X(s)\} \end{aligned}$$

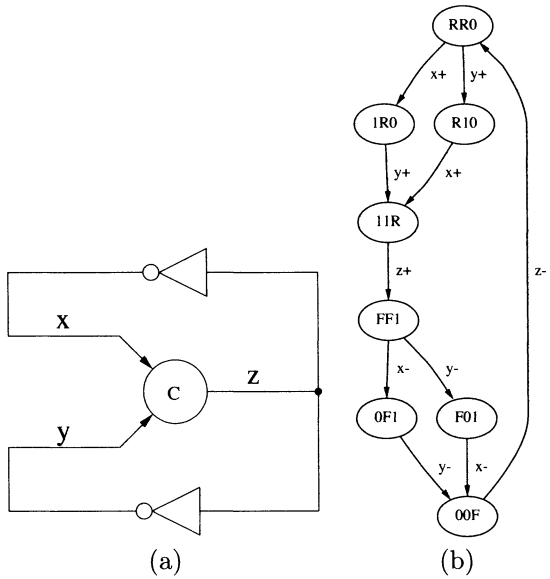

Fig. 6.4 (a) Semi-modular circuit. (b) State graph with state vector $\langle x, y, z \rangle$.

Recall that $X(s)$ is the set of signals that are excited in state s.

For each signal u, there are two sets of stable, or *quiescent, states.* The sets $QS(u+)$ and $QS(u-)$ are defined as follows:

$$\begin{aligned} QS(u+) &= \{s \in S \mid s(u) = 1 \wedge u \notin X(s)\} \\ QS(u-) &= \{s \in S \mid s(u) = 0 \wedge u \notin X(s)\} \end{aligned}$$

Example 6.1.7 Consider the SG shown in Figure 6.4. The signal y has the following four sets:

$$\begin{aligned} ES(y+) &= \{\langle RR0 \rangle, \langle 1R0 \rangle\} \\ ES(y-) &= \{\langle FF1 \rangle, \langle 0F1 \rangle\} \\ QS(y+) &= \{\langle R10 \rangle, \langle 11R \rangle\} \\ QS(y-) &= \{\langle F01 \rangle, \langle 00F \rangle\} \end{aligned}$$

An *excitation region* for signal u is a maximally connected subset of either $ES(u+)$ or $ES(u-)$. If it is a subset of $ES(u+)$, it is a *set region*, and it is denoted $ER(u+, k)$ where k indicates that it is the kth set region. Similarly, a *reset region* can be denoted $ER(u-, k)$.

The *switching region* for a transition $u*$, $SR(u*, k)$, is the set of states directly reachable through transition $u*$:

$$SR(u*, k) = \{s_j \in S \mid \exists s_i \in ER(u*, k).(s_i, u*, s_j) \in \delta\}$$

where "*" indicates either "+" for set regions or "−" for reset regions.

Example 6.1.8 Again consider the example SG shown in Figure 6.4. The signal y has the following excitation and switching regions:

$$\begin{aligned} ER(y+,1) &= \{\langle RR0\rangle, \langle 1R0\rangle\} \\ ER(y-,1) &= \{\langle FF1\rangle, \langle 0F1\rangle\} \\ SR(y+,1) &= \{\langle R10\rangle, \langle 11R\rangle\} \\ SR(y-,1) &= \{\langle F01\rangle, \langle 00F\rangle\} \end{aligned}$$

Another interesting subclass of speed-independent circuits are those which have *distributive* state graphs. A state graph is distributive if each excitation region has a unique *minimal state*. A minimal state for an excitation region, $ER(u*,k)$, is a state in $ER(u*,k)$, which cannot be directly reached by any other state in $ER(u*,k)$. More formally, a SG is distributive if

$$\forall ER(u*,k)\ .\ \exists \text{exactly one } s_j \in ER(u*,k)\ .\ \neg\exists s_i \in ER(u*,k)\ .\ (s_i,t,s_j) \in \delta$$

Example 6.1.9 The circuit in Figure 6.2 is not distributive, since for $ER(z+,1)$ there exists two minimal states, $\langle 1RR\rangle$ and $\langle R1R\rangle$. The circuit in Figure 6.4 is distributive.

Each cube in the implementation is composed of *trigger signals* and *context signals*. For an excitation region, a trigger signal is a signal whose firing can cause the circuit to enter the excitation region. The set of trigger signals for an excitation region $ER(u*,k)$ is

$$\begin{aligned} TS(u*,k) = \{v \in N \mid \exists s_i, s_j \in S.((s_i,t,s_j) \in \delta) \wedge (t = v+ \vee t = v-) \\ \wedge\ (s_i \notin ER(u*,k)) \wedge (s_j \in ER(u*,k))\} \end{aligned}$$

Any nontrigger signal which is stable throughout an entire excitation region may be used as a context signal. The set of context signals for an excitation region $ER(u*,k)$ is

$$CS(u*,k) = \{v_i \in N \mid v_i \notin TS(u*,k) \wedge \forall s_j, s_l \in ER(u*,k).s_j(i) = s_l(i)\}$$

Example 6.1.10 Once again consider the example SG shown in Figure 6.4. The excitation regions for signal y have the following trigger and context signal sets:

$$\begin{aligned} TS(y+,1) &= \{z\} \\ TS(y-,1) &= \{z\} \\ CS(y+,1) &= \{y\} \\ CS(y-,1) &= \{y\} \end{aligned}$$

Note that although the signal y is rising in $ER(y+,1)$, it is considered stable since once y does rise the circuit has left the excitation region. Recall that a state labeled with an R is actually at the fixed logical value of 0, but it is excited to change.

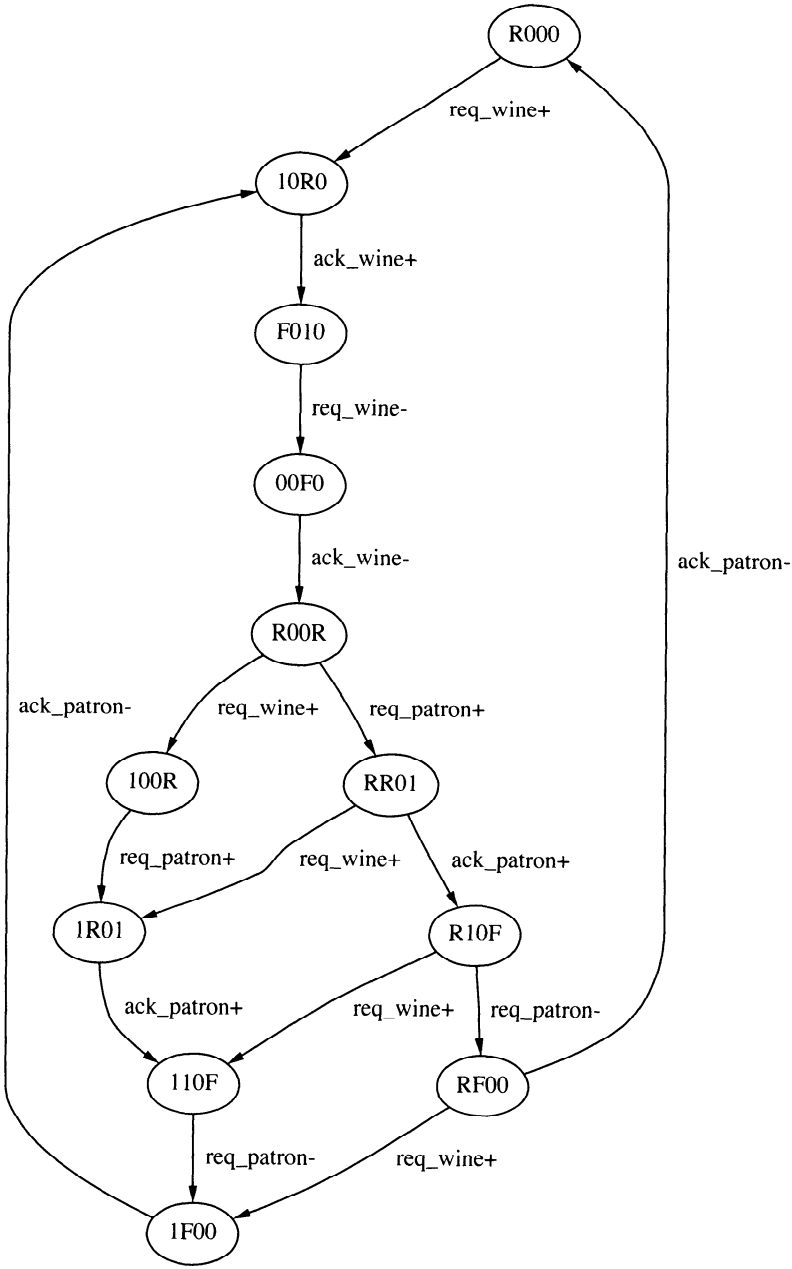

Fig. 6.5 State graph for the passive/active wine shop (statevector is $\langle req\_wine, ack\_patron, ack\_wine, req\_patron \rangle$).

Example 6.1.11 We conclude this section with a somewhat larger example. The SG for a passive/active wine shop is shown in Figure 6.5. The *ES*, *QS*, *ER*, *SR*, *TS*, and *CS* sets for the signal *req_patron* are shown below.

$$\begin{aligned}
ES(req_patron+) &= \{\langle R00R\rangle, \langle 100R\rangle\} \\
ES(req_patron-) &= \{\langle R10F\rangle, \langle 110F\rangle\} \\
QS(req_patron+) &= \{\langle RR01\rangle, \langle 1R01\rangle\} \\
QS(req_patron-) &= \{\langle RF00\rangle, \langle 1F00\rangle, \langle R000\rangle, \langle 10R0\rangle, \\
&\quad \langle F010\rangle, \langle 00F0\rangle\} \\
ER(req_patron+,1) &= \{\langle R00R\rangle, \langle 100R\rangle\} \\
ER(req_patron-,1) &= \{\langle R10F\rangle, \langle 110F\rangle\} \\
SR(req_patron+,1) &= \{\langle RR01\rangle, \langle 1R01\rangle\} \\
SR(req_patron-,1) &= \{\langle RF00\rangle, \langle 1F00\rangle\} \\
TS(req_patron+,1) &= \{ack_wine\} \\
TS(req_patron-,1) &= \{ack_patron\} \\
CS(req_patron+,1) &= \{ack_patron,\ req_patron\} \\
CS(req_patron-,1) &= \{ack_wine,\ req_patron\}
\end{aligned}$$

6.2 COMPLETE STATE CODING

Two states have *unique state codes* (USC) if they are labeled with different binary vectors. A SG has USC if all state pairs have USC. This is presented more formally below.

$$\begin{aligned}
USC(s_i, s_j) &\Leftrightarrow \lambda_S(s_i) \neq \lambda_S(s_j) \\
USC(S) &\Leftrightarrow \forall (s_i, s_j) \in S \times S \;.\; USC(s_i, s_j)
\end{aligned}$$

Two states have *complete state codes* (CSC) if they either have USC or do not have USC but do have the same output signals excited in each state. A SG has CSC if all state pairs have CSC. This is presented more formally below.

$$\begin{aligned}
CSC(s_i, s_j) &\Leftrightarrow USC(s_i, s_j) \ \vee\ X(s_i) \cap O = X(s_j) \cap O \\
CSC(S) &\Leftrightarrow \forall (s_i, s_j) \in S \times S \;.\; CSC(s_i, s_j)
\end{aligned}$$

A set of state pairs which violate the CSC property is defined as

$$CSCV(S) = \{(s_i, s_j) \in S \times S \mid \neg\, CSC(s_i, s_j)\}$$

Example 6.2.1 Consider the state graph shown in Figure 6.5. The states $\langle R000\rangle$ and $\langle R00R\rangle$ do not have USC since they share the same underlying state code $\langle 0000\rangle$. Similarly, the states $\langle 10R0\rangle$ and $\langle 100R\rangle$ do not have USC. Recall that the output signals for this circuit are *ack_wine* and *req_patron*. Therefore, states $\langle R000\rangle$ and $\langle R00R\rangle$ also do

not have CSC, since *req_patron* is stable low in the first state and excited to rise in the second. Similarly, states $\langle 10R0 \rangle$ and $\langle 100R \rangle$ do not have CSC due to differences in output enablings. If *req_patron* had been an input, the first pair would have satisfied the CSC property. The second still would not, since *ack_wine* is only excited in the first state of the pair.

When a SG does not have CSC, the implied value for some output signals cannot be determined by simply considering the values of the signal wires. This ambiguity leads to a state of confusion for the circuit. Note that if a SG does not have USC but has CSC, there is no problem since a circuit only needs to be synthesized for the output signals. To synthesize a circuit from a SG that does not have CSC, the specification must be modified. One possibility is to reshuffle the protocol as described in Chapter 3. In this section we describe a method for inserting state variables to solve the CSC problem.

6.2.1 Transition Points and Insertion Points

The insertion of a state signal into a circuit involves the addition of a rising and falling transition on the new signal. *Transition points* are a useful way of specifying where these transitions occur. A transition point is an ordered pair of sets of transitions, $TP = (t_s, t_e)$, where t_s is a set of *start transitions* and t_e is a set of *end transitions*. The transition point represents the location in the protocol in which a transition on a new state signal is to be inserted. In a graphical model such as a STG, each transition point represents a transition with incoming arcs from each of the transitions in t_s and with outgoing arcs to the transitions in t_e. An *insertion point* consists of an ordered pair of transition points, $IP = (TP_R, TP_F)$, where TP_R is for the rising transition and TP_F is for the falling transition.

It is necessary to determine in which states a transition of a new state signal can occur when inserted into the circuit using a given TP. The signal transition becomes excited after all transitions in t_s have occurred. Assume that $t \in t_s$; then we know that t has occurred when we enter the switching region for t. We know that all $t \in t_s$ have occurred when we are in the switching region for each t. Therefore, the transition on the new state signal becomes excited when the circuit enters the intersection of all the switching regions for each transition in t_s [i.e., $\cap_{t \in t_s} SR(t)$]. The transition on this new signal is guaranteed to have completed before any transition in t_e can occur. Therefore, once this transition becomes excited, it may remain excited in any subsequent states until a state is reached through a transition in t_e. We can now recursively define the set of states in which a new transition inserted into TP is excited.

$$S(\mathrm{TP}) \quad = \quad \{s_j \in S \mid s_j \in \cap_{t \in t_s} SR(t) \vee (\exists (s_i, t, s_j) \in \delta . s_i \in S(\mathrm{TP}) \wedge t \notin t_e)\}$$

Example 6.2.2 Consider the state graph shown in Figure 6.5 with $TP = (\{req\_patron+\}, \{req\_patron-\})$. SR($req\_patron+$) is $\langle RR01\rangle$ and $\langle 1R01\rangle$, so $S(TP)$ is seeded with these states. To find the rest of the states in $S(TP)$, we follow state transitions from these states until *req_patron−* occurs. For example, ($\langle RR01\rangle$, *ack_patron+*, $\langle R10F\rangle$) $\in \delta$, so $\langle R10F\rangle$ is in $S(TP)$. The transition ($\langle 1R01\rangle$, *ack_patron+*, $\langle 110F\rangle$) $\in \delta$, so $\langle 110F\rangle$ is also in $S(TP)$. However, the transitions leaving these two states involve *req_patron−*, so there are no more states in $S(TP)$. In summary, S($\{req\_patron+\}$, $\{req\_patron-\}$) is

$$\{\langle RR01\rangle, \langle 1R01\rangle, \langle R10F\rangle, \langle 110F\rangle\}$$

Theoretically, the set of all possible insertion points includes all combinations of transitions in t_s and t_e for the rising transition and all combinations of transitions in t_s and t_e for the falling transition. Thus, the upper bound on the number of possible insertion points is $2^{|T|^4}$. Fortunately, many of these insertion points can be quickly eliminated because they either never lead to a satisfactory solution of the CSC problem or the same solution is found using a different insertion point.

A transition point must satisfy the following three restrictions:

1. The start and end sets should be disjoint (i.e., $t_s \cap t_e = \emptyset$).
2. The end set should not include input transitions (i.e., $\forall t \in t_e \; . \; t \notin T_I$).
3. The start and end sets should include only concurrent transitions (i.e., $\forall t_1, t_2 \in t_s \; . \; t_1 \parallel t_2$ and $\forall t_1, t_2 \in t_e \; . \; t_1 \parallel t_2$).

The first requirement, that t_s and t_e be disjoint, simply eliminates unnecessary loops. The second requirement is necessary since the interface behavior is assumed to be fixed. In other words, we are not allowed to change the way the environment reacts to changes in output signals. In particular, if we allowed input transitions in t_e, this would force the environment to delay an input transition until a state signal changes. The third requirement is that all transitions in t_s and t_e be mutually concurrent. A transition point which contains transitions that are not concurrent describes the same behavior (i.e., the same set of states where the new signal transition is excited) as a transition point that satisfies these requirements.

Example 6.2.3 Again, consider the state graph shown in Figure 6.5. The transition point ($\{req\_patron+\}$, $\{req\_patron+\}$) violates the first requirement, and it is clearly not useful as it implies that the new signal transition is excited in all states. The transition point ($\{req\_patron+\}$, $\{ack\_patron+\}$) violates the second requirement, and it would force *ack_patron* to wait to see the state signal change before it could rise. This would require the interface behavior to be changed, which is not allowed. The transition point ($\{ack\_wine-, req\_patron+\}$, $\{req\_patron-\}$) violates the third requirement since *ack_wine−* is not concurrent with but rather precedes *req_patron+*. This transition point implies the same

set of states as ({*req_patron+*}, {*req_patron−*}). The transition point ({*req_patron+*}, {*ack_wine+*, *req_patron−*}) also violates the third requirement since *req_patron−* is not concurrent with *ack_wine+*, but rather, precedes it. This transition point also implies the same states as ({*req_patron+*}, {*req_patron−*}).

Some of the legal transition points are given below.

$$
\begin{aligned}
(\{ack\_wine+\} &, \{ack\_wine-\}) \\
(\{ack\_wine-\} &, \{ack\_wine+\}) \\
(\{req\_wine-\} &, \{ack\_wine-\}) \\
(\{req\_patron+\} &, \{req\_patron-\}) \\
(\{ack\_patron+\} &, \{req\_patron-\}) \\
(\{req\_wine+, req\_patron-\} &, \{ack\_wine+\}) \\
(\{req\_wine+, req\_patron-\} &, \{ack\_wine-\}) \\
(\{req\_wine+, ack\_patron-\} &, \{ack\_wine+\}) \\
(\{req\_wine+, ack\_patron-\} &, \{ack\_wine-\})
\end{aligned}
$$

Once two legal and useful TPs have been found, they are combined into an insertion point $IP = (TP_R, TP_F)$ and checked for compatibility. Two transition points are incompatible when either of the following is true:

$$
\begin{aligned}
TP_R(t_s) \cap TP_F(t_s) &\neq \emptyset \\
TP_R(t_e) \cap TP_F(t_e) &\neq \emptyset
\end{aligned}
$$

For a state graph to have consistent state assignment, a transition on a signal must be followed by an opposite transition before another transition of the same type can occur. An incompatible insertion point always creates an inconsistent state assignment.

Example 6.2.4 The transition points ({*ack_wine+*}, { *ack_wine−* }) and ({*req_wine+*, *req_patron−*}, {*ack_wine−*}) would not form a compatible insertion point since *ack_wine−* is in both $TP_R(t_e)$ and $TP_F(t_e)$.

6.2.2 State Graph Coloring

After finding all compatible insertion points, the next step is to determine the effect of inserting a state variable into each insertion point. This could be determined simply by inserting the state signal and rederiving the SG. This approach is unnecessarily time consuming and may produce a SG with an inconsistent state assignment. To address both of these problems, the original SG is partitioned into four subpartitions, corresponding to the states in which the new signal is rising, falling, stable high, and stable low.

Partitioning is accomplished by coloring each state in the original SG. First, all states in $S(TP_R)$ are colored as *rising*, which indicates that the state signal would be excited to rise in these states. The states in $S(TP_F)$ are colored as *falling*. If during the process of coloring *falling* states, a state is

found that has already been colored as *rising*, this insertion point leads to an inconsistent state assignment and must be discarded. Once both the rising and falling states have been colored, all states following those colored rising before reaching any colored falling are colored as *high*. Similarly, all states between those colored as falling and those colored as rising are colored as *low*. While coloring *high* or *low*, if a state to be colored is found to already have a color, then again the insertion point leads to an inconsistent state assignment.

Example 6.2.5 Consider the insertion point

$$\text{IP}((\{req\_patron+\}, \{req\_patron-\}), (\{ack\_wine-\}, \{ack\_wine+\}))$$

We would first color all states in S($\{req\_patron+\}$, $\{req\_patron-\}$) which we found previously to be $\{\langle RR01\rangle, \langle 1R01\rangle, \langle R10F\rangle, \langle 110F\rangle\}$ to be rising. Next, we would color all states in $S(\{ack\_wine-\}, \{ack\_wine+\})$ to be falling. However, this would result in each of the following states $\{\langle RR01\rangle, \langle 1R01\rangle, \langle R10F\rangle, \langle 110F\rangle\}$ to be colored both rising and falling. Therefore, this insertion point results in an inconsistent state assignment and is discarded.

Example 6.2.6 Consider the insertion point

$$\text{IP}((\{ack\_wine+\}, \{ack\_wine-\}), (\{req\_patron+\}, \{req\_patron-\}))$$

Coloring the SG would result in the following partition:

$$\begin{aligned} rising &= \{\langle F010\rangle, \langle 00F0\rangle\} \\ falling &= \{\langle RR01\rangle, \langle 1R01\rangle, \langle R10F\rangle, \langle 110F\rangle\} \\ high &= \{\langle R00R\rangle, \langle 100R\rangle\} \\ low &= \{\langle RF00\rangle, \langle 1F00\rangle, \langle R000\rangle, \langle 10R0\rangle\} \end{aligned}$$

This coloring is also shown in Figure 6.6. Notice that for this insertion point, the states $\langle R000\rangle$ and $\langle 10R0\rangle$ are colored as low and states $\langle R00R\rangle$ and $\langle 100R\rangle$ are colored as high. This means that in the first two states the new state signal is stable low and in the last two states the new state signal is stable high. The result is that these states, which previously had a CSC violation, no longer have a CSC violation since the new state signal disambiguates them.

6.2.3 Insertion Point Cost Function

After partitioning the SG determines that an insertion point leads to a consistent state assignment, the next step is to determine if the insertion point found is better than one found previously. The primary component of the cost function is the number of CSC violations which would remain after a state signal is inserted into a given IP. The number of remaining CSC violations for a given IP is determined by eliminating from CSCV any pair of violations in which one state is colored *high* while the other is colored *low*. Next, states that previously had a USC violation may now have a CSC violation due to the

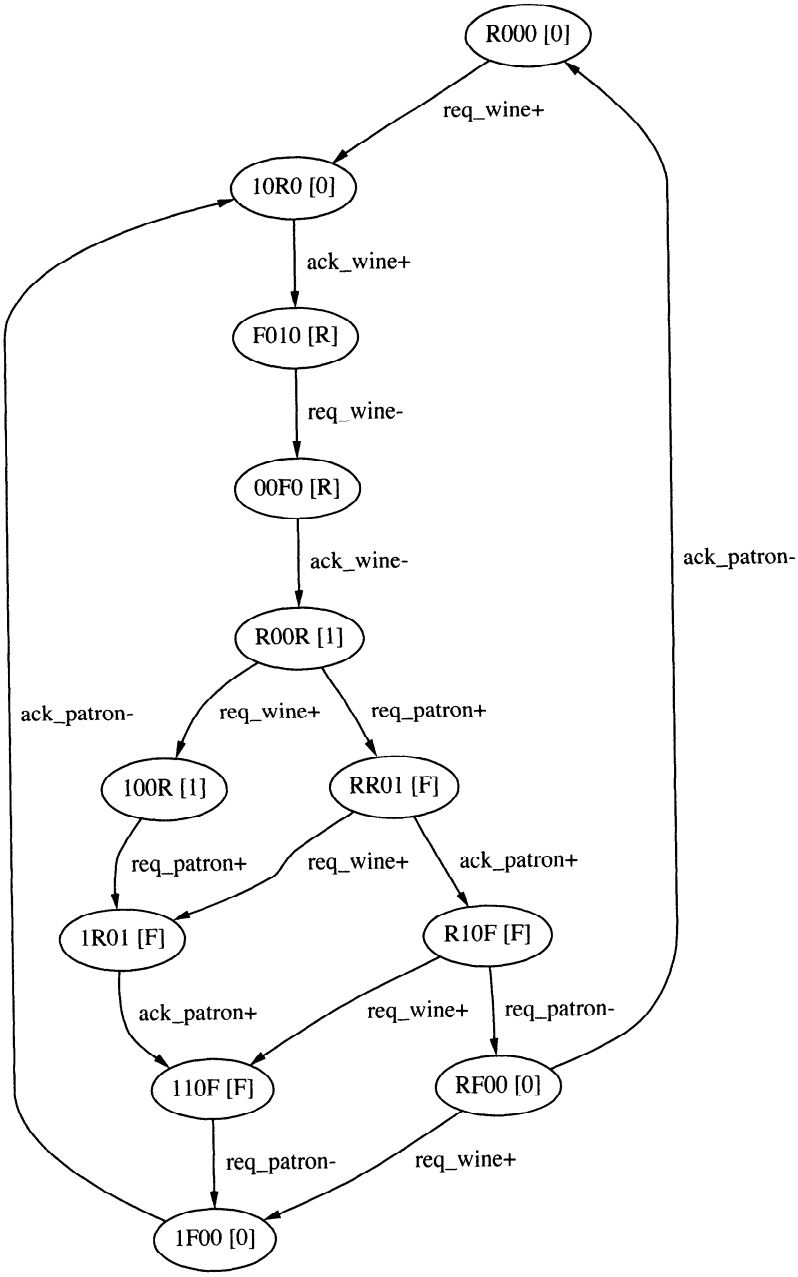

Fig. 6.6 Colored state graph for the passive/active wine shop (statevector is ⟨*req_wine*, *ack_patron*, *ack_wine*, *req_patron*⟩).

insertion of the state signal. In particular, for each pair of states with a USC violation (but not a CSC violation), if one is colored *rising* while the other is colored *low*, there is now a CSC violation since these states have different output enablings. Similarly, if one is colored *falling* and the other is colored *high*, there is also a new CSC violation. Each new CSC violation of the type just described must be added to the total remaining.

When two IPs leave the same number of CSC violations, a secondary cost function must be used. Each additional criterion is considered in order until the tie is broken. One possible additional cost function is to select the IP with the smallest sum, $|TP_R(t_e)| + |TP_F(t_e)|$, since it delays a smaller number of other transitions. Another is to select the IP with the smallest sum, $|TP_R(t_s)| + |TP_F(t_s)|$, since fewer enabling signals leads to simpler circuits.

6.2.4 State Signal Insertion

Once a good insertion point has been selected, the next step is to insert the state signal into the SG. This can be accomplished either by adding the transition to the higher-level representation, such as a STG or TEL structure, or by expanding the SG. If the STG is to be modified, arcs are added from each transition in t_s to the new state signal transition. Similarly, arcs are added from the new transition to each of the transitions in t_e. The same steps are followed for the reverse transition of the state signal. An initial marking for the new arcs must also be determined that preserves liveness and safety of the STG. After both transitions have been added to the STG, the state signal is assigned an initial value based on the coloring of the initial state. If the initial state is colored as *high* or *falling*, the initial value is high. Otherwise, the initial value is low. At this point, a new SG can be found.

Example 6.2.7 Again consider the insertion point

$$\mathrm{IP}((\{ack\_wine+\},\{ack\_wine-\}),(\{req\_patron+\},\{req\_patron-\}))$$

The STG for this insertion point is shown in Figure 6.7. The state graph for this STG is shown in Figure 6.8, and it has CSC.

Alternatively, the new SG can be found directly. Each state in the original state graph is extended to include one new signal value. If a state is colored *low*, the new signal is '0' in that state. If a state is colored *high*, the new signal is '1'. If a state is colored *rising*, it must be split into two new states, one in which the new signal is 'R' and another in which the new signal is '1'. Similarly, if a state is colored *falling*, it must be split into two new states, where one has the new signal as 'F' and the other has it as '0'.

Example 6.2.8 Compare the state graphs shown in Figures 6.6 and 6.8. Notice that all states colored with a stable value, 0 or 1, appear in the new state graph simply extended with that new variable. In the case of states colored with an unstable value, R or F, must be split to show the change in the state variable. For example, the state $\langle F010\rangle$ is

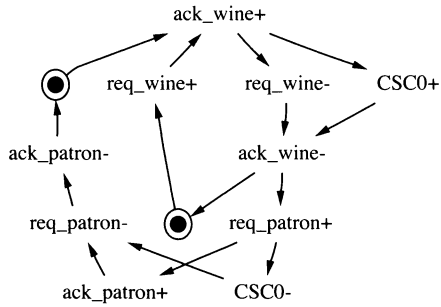

Fig. 6.7 STG for the passive/active wine shop which has CSC.

extended to $\langle F010R \rangle$ and a new state must be added in which the state signal changes, $\langle F0101 \rangle$. For the state $\langle 00F0 \rangle$, the enabling of *ack_wine* must be removed in the extended state since it is in the end set and does not become enabled until after *CSC0* changes. Therefore, the new extended state becomes $\langle 0010R \rangle$. A new state must also be added for after *CSC0* rises in which *ack_wine* is now enabled (i.e., $\langle 00F01 \rangle$). The states for the falling of the state signal are expanded similarly.

6.2.5 Algorithm for Solving CSC Violations

The algorithm for solving CSC violations is shown in Figure 6.9. It first checks if there are any CSC violations. If there are, it finds all legal transition points. Next, it considers each pair of transition points as a potential insertion point. For each legal insertion point, it colors the state graph. If the colored state graph is consistent and the cost of this insertion point is better than the best found so far, it records it. Finally, it inserts the new signal into the best insertion point found. Often, the insertion of a single state signal is not sufficient to eliminate all CSC violations. Therefore, after deriving the new SG, it may be necessary to solve CSC violations in the new SG. It does this by calling the CSC solver recursively and adds an additional state signal.

6.3 HAZARD-FREE LOGIC SYNTHESIS

After generating a SG with CSC, we apply a modified logic minimization procedure to obtain a hazard-free logic implementation. The modifications necessary are dependent upon the assumed technology. In this section we describe the necessary modifications for three potential technologies: complex gates, generalized C-elements, and basic gates. We conclude this section with a description of an extremely efficient logic minimization algorithm which can be applied to an important subclass of specifications.

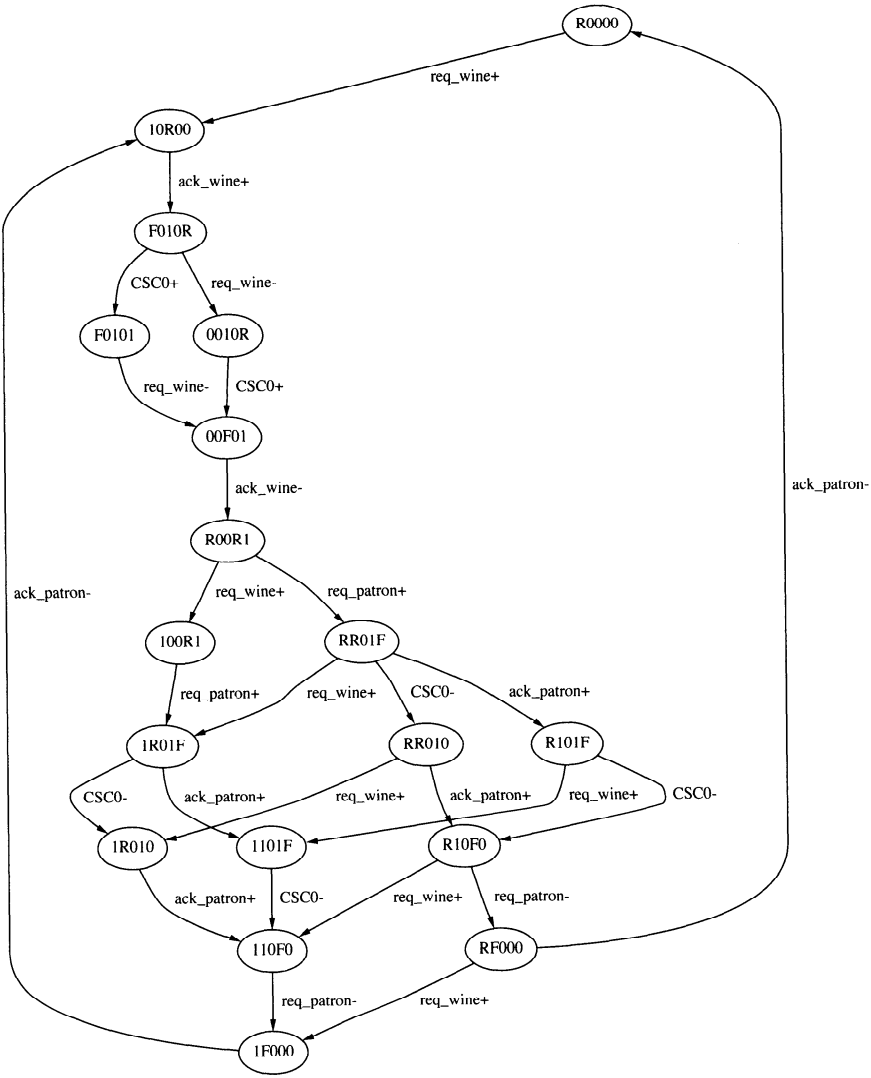

Fig. 6.8 State graph for the passive/active wine shop with CSC (state vector is $\langle req\_wine, ack\_patron, ack\_wine, req\_patron, CSC0 \rangle$).

```
csc_solver(SG) {
  CSCV = find_csc_violations(SG);
  if (|CSCV| = 0) return SG;        /* No CSC violations so return.*/
  best = |CSCV|;
  best_IP = (∅,∅);                  /* Initialize best insertion point.*/
  TP = find_all_transition_points(SG);
  foreach TP_R ∈ TP
    foreach TP_F ∈ TP
      if IP = (TP_R,TP_F) is legal then {
        CSG = color_state_graph(SG,TP_R,TP_F);
        CSCV = find_csc_violations(CSG);
        if (CSG is consistent) and ((|CSCV| < best) or
          ((|CSCV| = best) and (cost(IP) < cost(best_IP)))) then {
          best = |CSCV|;
          best_IP = (TP_R,TP_F);          /* Record new best IP.*/
        }
      }
  SG = insert_state_signal(SG,best_IP);
  SG = csc_solver(SG);              /* Add more signals, if needed.*/
  return SG;
}
```

Fig. 6.9 Algorithm for solving CSC violations.

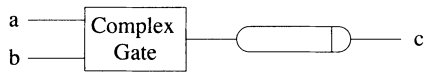

Fig. 6.10 Atomic gate model.

6.3.1 Atomic Gate Implementation

In the first synthesis method, we assume that each output is implemented using a single complex *atomic gate*. A gate is atomic when its delay is modeled by a single delay element connected to its output as depicted in Figure 6.10.

Using this model, after obtaining a SG with CSC, we can apply traditional logic minimization to find a logic implementation. The ON-set for a signal u is the set of all states in which u is either excited to rise [i.e., $ES(u+)$] or stable high [i.e., $QS(u+)$]. The OFF-set is the set of all states in which u is either excited to fall [i.e., $ES(u-)$] or stable low [i.e., $QS(u-)$]. The DC-set is the set of all unreachable states, or equivalently, those states not included in either the ON-set or OFF-set. In other words, the state space is partitioned for a signal u as follows:

$$\begin{aligned} \textit{ON-set} &= \{\lambda_S(s) \mid s \in (ES(u+) \cup QS(u+))\} \\ \textit{OFF-set} &= \{\lambda_S(s) \mid s \in (ES(u-) \cup QS(u-))\} \\ \textit{DC-set} &= \{0,1\}^{|N|} - (\textit{ON-set} \cup \textit{OFF-set}) \end{aligned}$$

We apply the recursive prime generation procedure described earlier to find all prime implicants. Finally, we set up and solve a covering problem to find the minimum number of primes that covers the minterms in the *ON-set*.

Example 6.3.1 Consider the SG with CSC shown in Figure 6.8. For the signal *ack_wine*, we would find the following sets:

$$\begin{aligned} \textit{ON-set} &= \{10000, 10100, 00100, 10101\} \\ \textit{OFF-set} &= \{00101, 00001, 10001, 00011, 10011, 01011, 00010, \\ &\quad 10010, 01010, 11010, 01000, 11000, 11011, 00000\} \\ \textit{DC-set} &= \{00110, 00111, 01001, 01100, 01101, 01110, 01111, \\ &\quad 10110, 10111, 11001, 11100, 11101, 11110, 11111\} \end{aligned}$$

The primes found for *ack_wine* are

$$P = \{\texttt{1-1--}, \texttt{-11--}, \texttt{--11-}, \texttt{--1-0}, \texttt{-1-01}, \texttt{10-00}\}$$

The constraint matrix for *ack_wine* is shown below.

	1-1--	-11--	--11-	--1-0	-1-01	10-00
10000	–	–	–	–	–	1
10100	1	–	–	1	–	1
00100	–	–	–	1	–	–
10101	1	–	–	–	–	–

The primes `1-1--`,`--1-0`, and `10-00` are essential and cover the entire ON-set. In a similar fashion, we can find the logic implementations for *req_patron* and *CSC0*. The final circuit is shown in Figure 6.11(a).

The circuit in Figure 6.11(a) is hazard-free if all the delay is modeled as being at the output of the final gate in each signal network (i.e., the output of the OR gates). Unfortunately, if this circuit is mapped to basic gates and the delays of these gates are considered individually, the implementation may be hazardous. Consider the sequence of states shown in Figure 6.11(b). After *req_wine* goes high, *u2* goes high, causing *ack_wine* to go high. Assume that the gate feeding *u3* is slow to rise. At this point, if *req_wine* goes low, *u2* could go low, causing *ack_wine* to become excited to fall. Now, if *u3* finally rises, *ack_wine* would become excited to rise. The result is a $1 \rightarrow 0 \rightarrow 1$ glitch on the signal *ack_wine*.

6.3.2 Generalized C-Element Implementation

Another implementation strategy is to use generalized C-elements (gC) (see Figure 5.45). Using the gC approach, two minimization problems must now be solved for each signal u. The first implements the set of the function [i.e., *set*(u)], and the second implements the reset [i.e., *reset*(u)]. To implement *set*(u), the ON-set is only the states in which u is excited to rise. The OFF-set is again the states in which u is excited to fall or is stable low. The DC-set now includes the states in which u is stable high as well as the unreachable

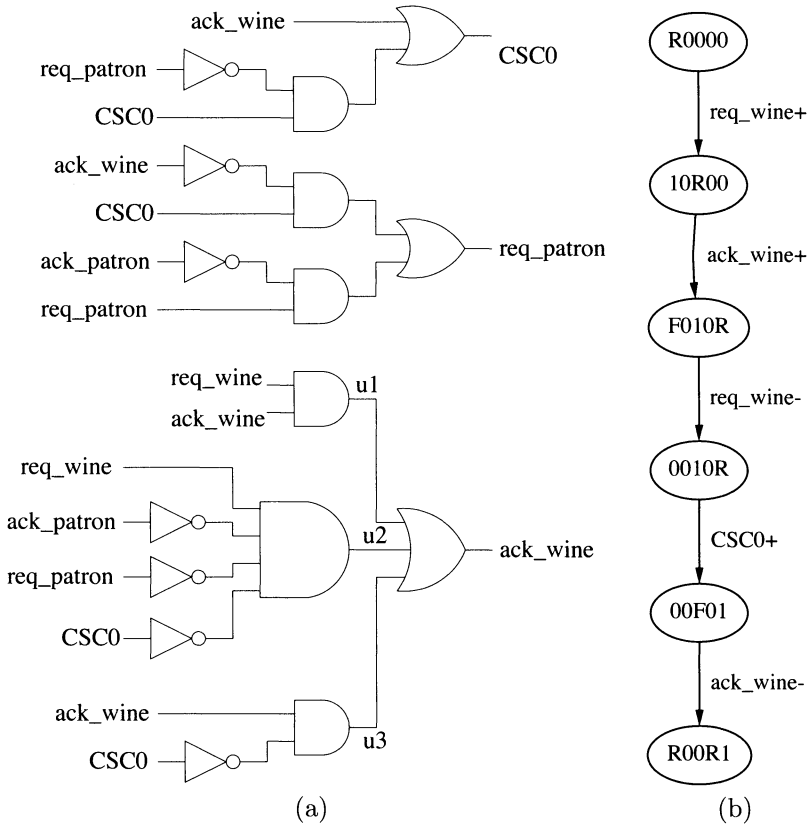

Fig. 6.11 (a) Atomic gate implementation of the passive/active shop. (b) Sequence of states leading to a gate-level hazard.

states. The stable high states are don't cares because once a gC is set, its feedback holds its state. In other words, we partition the state space as follows:

$$\begin{aligned} \textit{ON-set} &= \{\lambda_S(s) \mid s \in (ES(u+)\} \\ \textit{OFF-set} &= \{\lambda_S(s) \mid s \in (ES(u-) \cup QS(u-))\} \\ \textit{DC-set} &= \{0,1\}^{|N|} - (\textit{ON-set} \cup \textit{OFF-set}) \end{aligned}$$

To implement *reset*(u), the ON-set is the set of states in which u is excited to fall, and the OFF-set is the set of states in which u is either rising or high. The DC-set includes both the unreachable states and the states in which u is low. In other words, the partition becomes

$$\begin{aligned} \textit{ON-set} &= \{\lambda_S(s) \mid s \in (ES(u-)\} \\ \textit{OFF-set} &= \{\lambda_S(s) \mid s \in (ES(u+) \cup QS(u+))\} \\ \textit{DC-set} &= \{0,1\}^{|N|} - (\textit{ON-set} \cup \textit{OFF-set}) \end{aligned}$$

We can now apply standard methods to find a minimum number of primes to implement the set and reset functions.

Example 6.3.2 Again consider the SG with CSC shown in Figure 6.8. For *set(ack_wine)*, we would find the following sets:

$$\begin{aligned} \textit{ON-set} &= \{10000\} \\ \textit{OFF-set} &= \{00101, 00001, 10001, 00011, 10011, 01011, 00010, \\ &\quad 10010, 01010, 11010, 01000, 11000, 11011, 00000\} \\ \textit{DC-set} &= \{00110, 00111, 01001, 01100, 01101, 01110, 01111, \\ &\quad 10110, 10111, 11001, 11100, 11101, 11110, 11111, \\ &\quad 10100, 00100, 10101\} \end{aligned}$$

The primes for *ack_wine* are again found to be

$$P = \{\texttt{1-1--}, \texttt{-11--}, \texttt{--11-}, \texttt{--1-0}, \texttt{-1-01}, \texttt{10-00}\}$$

Only one prime, however, is needed to cover the ON-set: `10-00`. For the reset function, we also only need one prime `0---1`. In a similar fashion, we can find the logic implementations for the set and reset functions for *req_patron* and *CSC0*. The final gC circuit is shown in Figure 6.12.

Consider the sequence of states shown again in Figure 6.12(b). Again, *req_wine* rising causes *ack_wine* to rise. After *ack_wine* rises, *req_wine* is allowed to fall. There is no longer a potential for a hazard since the feedback in the gC implementation holds *ack_wine* stable high until *CSC0* rises at which point *ack_wine* is supposed to fall.

When the set function for a signal u, *set(u)*, is on in all states in which u should be rising or high, the state holding element can be removed. The implementation for u is simply equal to the logic for *set(u)*. Similarly, if *reset(u)* is on in all states in which u should be falling or low, the signal u can be implemented with $\overline{reset(u)}$. This process is called *combinational optimization.*

Example 6.3.3 Consider the SG shown in Figure 6.13(a). Using the generalized C-element implementation approach, we derive the circuit shown in Figure 6.13(b). The set of states in which signal d is rising or stable high is $110R$ and $11R1$ (note that in state $111F$, signal d is excited to fall). The logic for the set function, $ab\bar{c}$, evaluates to 1 in both of these states. Therefore, there is no need for a state-holding device, so we can use the circuit shown in Figure 6.13(c) for signal d instead.

If we use AND gates (with inverted inputs) and C-elements instead of gCs to implement the circuit shown in Figure 6.12, it would still be hazard-free. In general, however, this is not the case. Consider the SG and circuit shown in Figure 6.13. If we implement the circuit for signal c, using a two-input AND (with complemented first terminal), two-input OR, an inverter, and a C-element, this circuit is not hazard-free. Consider a sequence of states

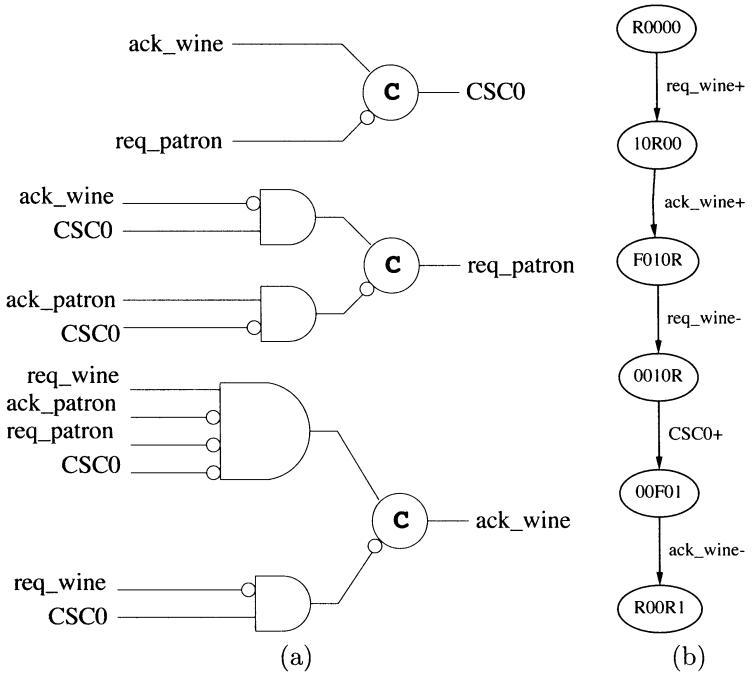

Fig. 6.12 (a) Generalized C-element implementation of passive/active shop. (b) Sequence of states leading to a gate-level hazard for the atomic gate circuit.

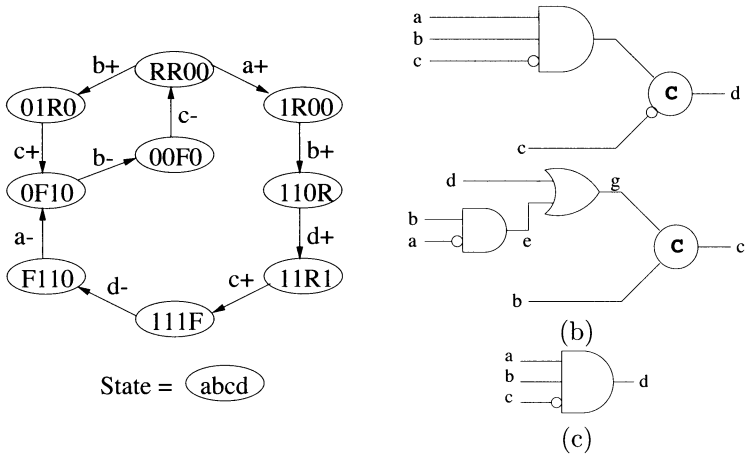

Fig. 6.13 (a) SG for small example. (b) Generalized C-element implementation. (c) Optimized combinational implementation of signal d.

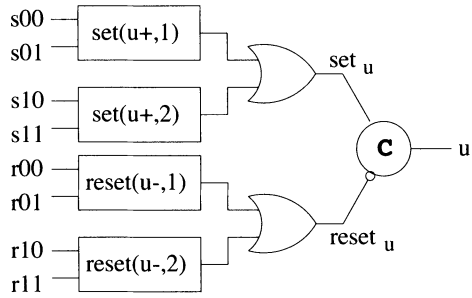

Fig. 6.14 Standard C-implementation.

beginning in state $\langle F110 \rangle$. Initially, signal e (the output of the two-input AND gate) is low, but after a goes low it becomes excited to rise. Let us assume that this gate is slow and e does not rise immediately. Next, b goes low, which disables e causing a hazard. The result can manifest in many ways. It may not cause any problem. It may cause c to fall slower. If c starts to fall and the glitch on e propagates to g causing g to rise, then c may stop falling and actually be restored back to its high value by the feedback in the C-element. Finally, if c falls and a and b rise before the glitch on e propagates to g, the glitch could cause c to turn on prematurely (i.e., before d has risen).

6.3.3 Standard C-Implementation

To avoid the hazard concerns discussed above, we could modify our logic minimization procedure to produce a gate-level hazard-free implementation called a *standard C-implementation.* The general structure of the standard C-implementation is shown in Figure 6.14. While the structure is similar to the gC-implementation, the method in which it is designed is quite different. First, each *region function* [i.e., $set(u+,k)$ or $reset(u-,k)$] implements a single (or possibly a set of) excitation region(s) for the signal u. In the gC-implementation, an excitation region can be implemented by multiple product terms. Second, each region function turns on only when it enters a state in its excitation region, turns off monotonically sometime after the signal u changes, and must stay off until the excitation region is entered again. To guarantee this behavior, each region function must satisfy certain correctness constraints, leading to a modified logic minimization procedure.

Each region function is implemented using a single atomic gate, corresponding to a *cover* of an excitation region. The cover of a set region $C(u+,k)$ [or a reset region $C(u-,k)$] is a set of states for which the corresponding region function in the implementation evaluates to one. While a single region function can be used to implement multiple excitation regions, we first present a method in which each region function only implements a single excitation region. Later, we extend the method to allow gate sharing.

For a cover to produce a gate-level hazard-free implementation, it must satisfy certain *correctness constraints*. The idea behind these constraints is that each region function can only change when it is needed to actively drive the output signal to change. Consider a region function for a set region. This gate turns on when the circuit enters a state in the set region. When the region function changes to 1, it excites the OR gate, which in turn excites the C-element (assuming that the reset network is low) to set u to 1. Only after u has risen can the region function be excited to fall. The region function then must fall monotonically. The signal u will not be able to fall until the region function has fallen and the OR gate for the set network has fallen. Once the region function falls, it is not allowed to be excited again until the circuit again enters a state in the corresponding set region. To guarantee this behavior, a correct cover must satisfy a *covering* and an *entrance constraint*.

First, a correct cover needs to satisfy a covering constraint which states that the reachable states in the cover must include the entire excitation region but must not include any states outside the union of the excitation region and associated quiescent states, that is,

$$ER(u*,k) \subseteq [C(u*,k) \cap S] \subseteq [ER(u*,k) \cup QS(u*)]$$

Second, the cover of each excitation region must also satisfy an entrance constraint which states that the cover must only be entered through excitation region states:

$$[(s_i,t,s_j) \in \delta \wedge s_i \notin C(u*,k) \wedge s_j \in C(u*,k)] \Rightarrow s_j \in ER(u*,k)$$

As the following theorem states, if all covers satisfy these two constraints, the resulting standard C-implementation is correct.

Theorem 6.4 (Beerel, 1998) *If for all outputs $u \in O$ all region function covers $C(u*,k)$ satisfy the covering and entrance constraints the standard C-implementation is correct (i.e., complex-gate equivalent and hazard-free).*

The goal of logic minimization is now to find an optimal sum-of-products function for each region function that satisfies the definition of a correct cover given above. An implicant of an excitation region is a product that may be part of a correct cover. In other words, a product c is an implicant of an excitation region $ER(u*,k)$ if the set of reachable states covered by c is a subset of the states in the union of the excitation region and associated quiescent states, that is,

$$[c \cap S] \subseteq [ER(u*,k) \cup QS(u*)]$$

For each set region $ER(u+,k)$, the ON-set is those states in $ER(u+,k)$. The OFF-set includes not only the states in which u is falling or low, but also the states outside this excitation region where u is rising. This additional restriction is necessary to make sure that a region function can only turn on

in its excitation region, and it will not glitch on in another excitation region for the same signal. More formally, we partition the state space as follows:

$$\begin{aligned} \textit{ON-set} &= \{\lambda_S(s) \mid s \in (ER(u+,k)\} \\ \textit{OFF-set} &= \{\lambda_S(s) \mid s \in (ES(u-) \cup QS(u-)) \cup (ES(u+) - ER(u+,k))\} \\ \textit{DC-set} &= \{0,1\}^{|N|} - (\textit{ON-set} \cup \textit{OFF-set}) \end{aligned}$$

The ON-set, OFF-set, and DC-set for a reset region $ER(u-,k)$ can be defined similarly:

$$\begin{aligned} \textit{ON-set} &= \{\lambda_S(s) \mid s \in (ER(u-,k)\} \\ \textit{OFF-set} &= \{\lambda_S(s) \mid s \in (ES(u+) \cup QS(u+)) \cup (ES(u-) - ER(u-,k))\} \\ \textit{DC-set} &= \{0,1\}^{|N|} - (\textit{ON-set} \cup \textit{OFF-set}) \end{aligned}$$

The prime implicants can again be found using standard techniques.

Example 6.3.4 Consider again the SG shown in Figure 6.13(a). There are two set regions for c: $ER(c+,1) = 01R0$ and $ER(c+,2) = 11R1$. Let's examine the implementation of $ER(c+,1)$. For this excitation region, we find the following partition of the state space:

$$\begin{aligned} \textit{ON-set} &= \{0100\} \\ \textit{OFF-set} &= \{0000, 1000, 0010, 1100, 1101\} \\ \textit{DC-set} &= \{0001, 0011, 0101, 0110, 0111, \\ &\quad 1001, 1010, 1011, 1110, 1111\} \end{aligned}$$

The primes found are as follows:

$$P = \{\texttt{01--}, \texttt{1-1-}, \texttt{-11-}, \texttt{0--1}, \texttt{-0-1}, \texttt{--11}\}$$

The entrance constraint creates a set of *implied states* for each implicant c [denoted $IS(c)$]. An implied state of an implicant c is a state that is not covered by c but due to the entrance constraint must be covered if the implicant is to be part of the cover. In other words, a state s is an implied state of an implicant c for the excitation region $ER(u*,k)$ if it is not covered by c, and s is a predecessor of a state that is both covered by c and not in the excitation region. This means that the product c becomes excited in a quiescent state instead of an excitation region state. If there does not exist some other product in the cover which contains this implied state, the cover violates the entrance constraint. More formally, the set of implied states for an implicant c is defined as follows:

$$IS(c) = \{s_i \mid s_i \notin c \wedge \exists s_j \,.\, (s_i, t, s_j) \in \delta \wedge (s_j \in c) \wedge (s_j \notin ER(u*,k))\}$$

An implicant may have implied states that are outside the excitation region and the corresponding quiescent states. Therefore, these implied states may not be covered by any other implicant. If this implicant is the only prime

implicant which covers some excitation region state, the covering problem cannot be solved using only prime implicants.

Example 6.3.5 Consider the prime implicant 01-- which is the only prime that covers the ON-set (i.e., the state $01R0$). This implicant can be entered through the transition $(F110, a-, 0F10)$. However, the state $0F10$ is not in $ER(c+, 1)$. Therefore, this would be an entrance violation, so the state $F110$ is an implied state for this implicant. This state must be covered by some other prime in the cover to satisfy the entrance constraint. There are two primes that cover this state: `1-1-` and `-11-`. If we include either of these primes, the cover can be entered through the transition $(11R1, c+, 111F)$, so $11R1$ is an implied state for these two primes. However, the state $11R1$ is in the OFF-set since it is part of another excitation region for c [i.e., $ER(c+, 2)$]. Therefore, we cannot include either of these primes in the cover, since we cannot cover their implied states. Therefore, no correct cover exists using only prime implicants.

To address this problem, we must introduce the notion of *candidate implicants*. An implicant is a candidate implicant if there exists no other implicant which properly contains it and has a subset of the implied states. In other words, c_i is a candidate implicant if there *does not exist* an implicant c_j that satisfies the following two conditions:

$$\begin{aligned} c_j &\supset c_i \\ IS(c_j) &\subseteq IS(c_i) \end{aligned}$$

Prime implicants are always candidate implicants, but not all candidate implicants are prime. An optimal cover can always be found using only candidate implicants.

Theorem 6.5 (Beerel, 1998) *An optimal cover of a region function always exists and consists of only candidate implicants.*

We find all candidate implicants using the algorithm in Figure 6.15. This algorithm is similar to the *prime_compatibles* algorithm from Chapter 5. The only real differences are that it checks implied states instead of class sets and uses the function *lit_extend* (instead of *max_subsets*) to find all implicants with one more literal than the given prime.

Example 6.3.6 Returning to our example, we would seed the list of candidate implicants with the primes found earlier. They are all of size 2. Let us first consider `01--`. As stated earlier, the implied state for this prime is $F110$. Since there is an implied state, we must consider extending this prime with an additional literal. The implicant `010-` has no implied states, and it is a subset of no other candidate implicant with no implied states, so it is added to the list of candidate implicants. The implicants `011-` and $0F10$ have $F110$ as an implied state, so they are not candidate implicants. Finally, 0111 has no implied states, but

```
candidate_implicants(SG, P) {
  done = ∅;                          /* Initialize already computed set.*/
  for (k = |largest(P)|; k ≥ 1; k − −) { /* Loop largest to smallest.*/
    foreach (q ∈ P; |q| = k) enqueue(C, q)   /* Queue all of size k.*/
    foreach (c ∈ C; |c| = k) {     /* Consider candidates of size k.*/
      if (IS(SG, c) = ∅) then continue          /* If empty, skip.*/
      foreach (s ∈ lit_extend(c)) {   /* Check extensions by 1 lit.*/
        if (s ∈ done) then continue          /* If computed, skip.*/
        Γ_s = IS(SG, s)          /* Find extension's implied states.*/
        prime = true                    /* Initialize prime as true.*/
        foreach (q ∈ C; |q| ≥ k) {       /* Check larger candidates.*/
          if (s ⊂ q) then {    /* If contained in prime, check it.*/
            Γ_q = IS(SG, q)              /* Compute implied states.*/
            if (Γ_s ⊇ Γ_q) then {      /* If smaller, not candidate.*/
              prime = false;
              break
            }
          }
        }
        if (prime = 1) then enqueue(C, s)   /* If prime, queue it.*/
        done = done ∪ {s}                    /* Mark as computed.*/
      }
    }
  }
  return(C);                        /* Return candidate implicants.*/
}
```

Fig. 6.15 Algorithm to find candidate implicants.

it is a subset of `0--1` which also has no implied states, so it is also not a candidate implicant. Next, we consider extending the prime `1-1-` since it has implied state $11R1$. The implicant `101-` has no implied states, so it is a candidate implicant. This is the last candidate implicant found, and the complete set of candidate implicants is:

$$\{\texttt{01--}, \texttt{010-}, \texttt{1-1-}, \texttt{101-}, \texttt{-11-}, \texttt{0--1}, \texttt{-0-1}, \texttt{--11}\}$$

We can now formulate a covering problem by introducing a Boolean variable x_i for each candidate implicant c_i. The variable $x_i = 1$ when the candidate implicant is included in the cover and 0 otherwise. Using these variables, we can construct a product-of-sums representation of the covering and entrance constraints.

First, a *covering clause* is constructed for each state s in the excitation region. Each clause consists of a disjunction of candidate implicants that cover s. More formally,

$$\bigvee_{i:s \in c_i} x_i$$

To satisfy the covering clause for each state s in $ER(u*, k)$, at least one x_i must be set to 1. This means that for each excitation region state s, there must be an implicant chosen that includes it in the cover. The set of covering clauses for an excitation region guarantees that all excitation region states are covered. Since candidate implicants are not allowed to include states outside the excitation region and corresponding quiescent states, the cover is guaranteed to satisfy the covering constraint.

Example 6.3.7 For our example there is only one excitation region state, 01R0, which is included only in candidate implicants c_1 (`01--`) and c_2 (`010-`), so we get the following covering clause:

$$(x_1 + x_2)$$

For each candidate implicant c_i, a *closure clause* is constructed for each of its implied states $s \in IS(c_i)$. Each closure clause represents an implication which states that if a candidate implicant is included in the cover, its implied states must also be included in some other implicant in the cover. This can be expressed formally as follows:

$$\overline{x_i} \vee \bigvee_{j:s \in c_j} x_j$$

The closure clauses ensure that the cover satisfies the entrance constraint.

Example 6.3.8 The candidate implicant c_1 (`01--`) has implied state 0F10. This state is included in the implicants c_3 (`1-1-`) and c_5 (`-11-`). Therefore, we get the following closure clause:

$$(\overline{x_1} + x_3 + x_5)$$

The complete formulation is

$$(x_1 + x_2)(\overline{x_1} + x_3 + x_5)\overline{x_3}\ \overline{x_5}\ \overline{x_8}$$

Our goal now is to find an assignment of the x_i variables that satisfies the function with the minimum cost where the cost is the number of implicants. Since there are negated variables, the covering problem is binate. To solve the binate covering problem, we again construct a constraint matrix to represent the product-of-sums described above. The matrix has one row for each clause and one column for each candidate implicant. The rows can be divided into a *covering section* and a *closure section*, corresponding to the covering and closure clauses. In the covering section, for each excitation region state s, a row exists containing a 1 in every column, corresponding to a candidate implicant that includes s. In the closure section, for each implied state s of each candidate implicant c_i, a row exists containing a 0 in the column corresponding to c_i and a 1 in each column corresponding to a candidate implicant c_j that covers the implied state s.

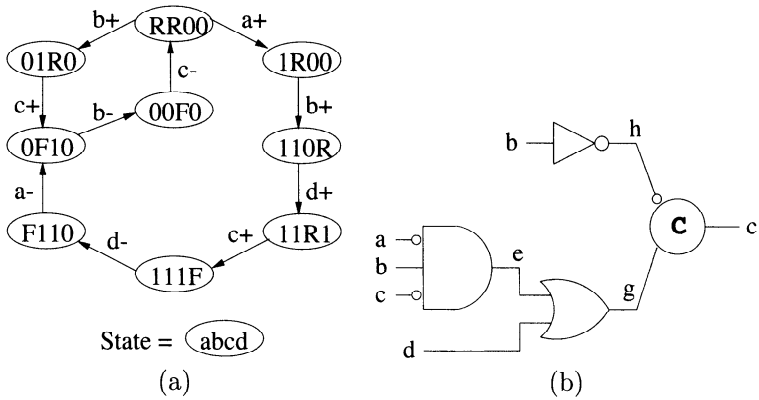

Fig. 6.16 (a) SG for small example. (b) Circuit found using basic gate approach.

Example 6.3.9 The constraint matrix for $ER(c+, 1)$ is depicted below.

	01--	010-	1-1-	101-	-11-	0--1	-0-1	--11
1	1	1	–	–	–	–	–	–
2	0	–	1	–	1	–	–	–
3	–	–	0	–	–	–	–	–
4	–	–	–	–	0	–	–	–
5	–	–	–	–	–	–	–	0

Rows 3, 4, and 5 are essential making the candidate implicants 1-1-, -11-, and --11 unacceptable, so these rows and the corresponding columns must be removed to produce the following matrix:

	01--	010-	101-	0--1	-0-1
1	1	1	–	–	–
2	0	–	–	–	–

Now, row 2 is essential making the implicant 01-- unacceptable, resulting in the following matrix:

	010-	101-	0--1	-0-1
1	1	–	–	–

Now, row 1 is essential, making the implicant 010- essential. Therefore, the cover includes only the implicant 010- (i.e., $\bar{a}b\bar{c}$). Note that this matrix can only be solved by selecting an implicant that is not prime. The resulting circuit implementation is shown in Figure 6.16. Consider again the sequence of states that led to a hazard beginning in state $F110$. When a falls, the signal e is not excited since c is high. Therefore, the hazard described earlier has been removed.

Again, we can apply a combinational optimization to the result. For standard C-implementations, we can remove the state-holding element when ei-

ther the set of covers for the set function for a signal u include all states where u is rising or high [i.e., $\bigcup_k C(u+,k) \supseteq ES(u+) \cup QS(u+)$], or the covers for the reset function include all states where u is falling or low [i.e., $\bigcup_k C(u-,k) \supseteq ES(u-) \cup QS(u-)$].

Another optimization is to allow a single gate to implement multiple excitation regions. The procedure finds a gate that covers each excitation region using modified correctness constraints. The covering constraint is modified to allow the cover to include states from other excitation regions, that is,

$$ER(u*,k) \subseteq [C(u*,k) \cap S] \subseteq \left[\bigcup_l ER(u*,l) \cup QS(u*)\right]$$

The entrance constraint must also be modified to allow the cover to be entered from any corresponding excitation region state:

$$[(s_i,t,s_j) \in \delta \wedge s_i \notin C(u*,k) \wedge s_j \in C(u*,k)] \Rightarrow s' \in \bigcup_l ER(u*,l)$$

An additional constraint is also now necessary to guarantee that a cover either includes an entire excitation region or none of it:

$$ER(u*,l) \not\subseteq C(u*,k) \Rightarrow ER(u*,l) \cap C(u*,k) = \emptyset$$

Example 6.3.10 Consider the state graph shown in Figure 6.17(a). The signal c has two set regions: $ER(c+,1) = 10R$ and $ER(c+,2) = 11R$. Using the earlier constraints, the primes are found to be

$$\begin{aligned} P(c+,1) &= \{\texttt{10-},\texttt{1-1},\texttt{-11}\} \\ P(c+,2) &= \{\texttt{11-},\texttt{1-1},\texttt{-11}\} \end{aligned}$$

The cover for $ER(c+,1)$ is the prime `10-`, since it covers the excitation region state $10R$ and has no implied states. For $ER(c+,2)$, the prime `11-` has implied state $FR1$. This implied state can be covered by `1-1`, but this prime has implied state $10R$ which is in the offset. Therefore, the prime `11-` must be expanded to `110` which is the final solution. Therefore, the set function for c is $a\bar{b} + ab\bar{c}$ [see Figure 6.17(b)].

Using the new constraints that allow gate sharing, the primes are now found to be

$$\begin{aligned} P(c+,1) &= \{\texttt{1--},\texttt{-11}\} \\ P(c+,2) &= \{\texttt{1--},\texttt{-11}\} \end{aligned}$$

With the new entrance constraint, a transition is allowed to enter a prime through any excitation region state. Therefore, the prime `1--` has no implied states and is a minimal cover. Therefore, using the new constraints, the set function for c is simply a [see Figure 6.17(c)].

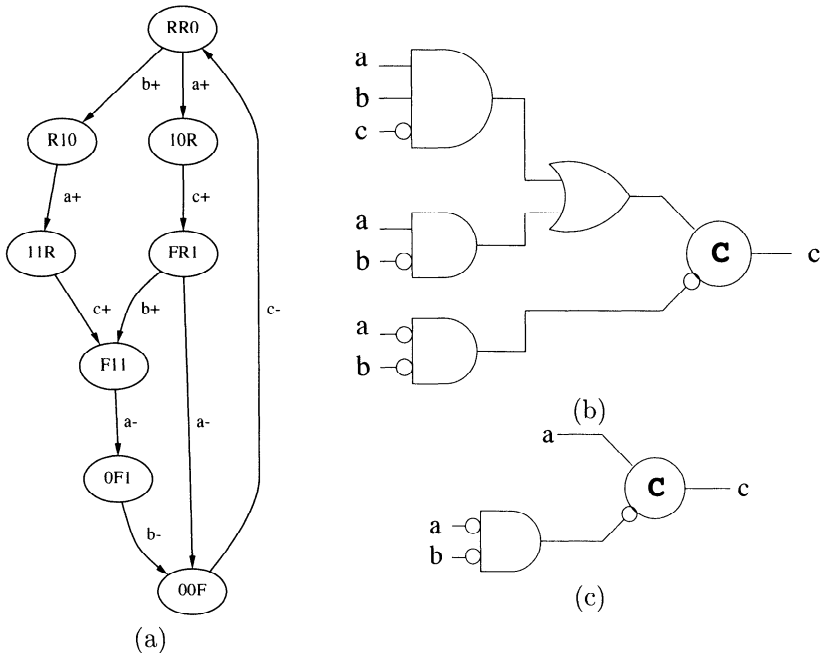

Fig. 6.17 (a) SG for gate-sharing example. (b) Original circuit without gate sharing. (c) Final circuit with gate sharing.

6.3.4 The Single-Cube Algorithm

The foregoing algorithms for logic minimization are often more general than necessary since many region functions can be implemented with a single product, or cube. This subsection presents a more efficient algorithm, shown in Figure 6.18, that finds an optimal single-cube cover, if one exists.

For a single-cube cover to hazard-freely implement a region function, all literals in the cube must correspond to signals that are *stable* throughout the excitation region. Otherwise, the single-cube cover would not cover all excitation region states. When a single-cube cover exists, an excitation region $ER(u*, k)$ can be sufficiently approximated using an *excitation cube* [denoted $EC(u*, k)$] which is the supercube of the states in the excitation region and is defined on each signal v as follows:

$$EC(u*, k)(v) \equiv \begin{cases} 0 & \text{if } \forall s \in ER(u*, k) \;.\; s(v) = 0 \\ 1 & \text{if } \forall s \in ER(u*, k) \;.\; s(v) = 1 \\ - & \text{otherwise} \end{cases}$$

If a signal has a value of 0 or 1 in the excitation cube, the signal can be used in the cube implementing the region. The set of states implicitly represented by

```
single_cube(SG,technology) {
  foreach u ∈ O {                    /* Consider each output signal.*/
    EC =find_excitation_cubes(SG);
    foreach EC(u*,k) ∈ EC {          /* Find cover for each EC.*/
      TC(u*,k) =find_trigger_cube(SG, EC(u*,k));
      CS(u*,k) =find_context_signals(SG, EC(u*,k), TC(u*,k));
      V(u*,k) =find_violations(SG, EC(u*,k), TC(u*,k),technology);
      CC =build_cover_table(CS(u*,k), V(u*,k));
      C(u*,k) =solve_cover_table(CC, TC(u*,k));
    }
    solution(u) = optimize_logic(C);  /* Combo opt, gate sharing.*/
  }
  return solution;
}
```

Fig. 6.18 Single-cube algorithm.

the excitation cube is always a superset of the set of excitation region states [i.e., $EC(u*,k) \supseteq ER(u*,k)$].

The set of trigger signals for an excitation region $ER(u*,k)$ can also be represented with a cube called a *trigger cube* $TC(u*,v)$, defined as follows for each signal v:

$$TC(u*,k)(v) \equiv \begin{cases} s_j(v) & \text{If } \exists(s_i,t,s_j) \in \delta \;.\; (t = v+ \vee t = v-) \wedge \\ & (s_i \notin EC(u*,k)) \wedge (s_j \in EC(u*,k)) \\ - & \text{otherwise} \end{cases}$$

The single-cube algorithm requires the cover of each excitation region to contain all its trigger signals [i.e., $C(u*,k) \subseteq TC(u*,k)$]. Since only stable signals can be included, a necessary condition for our algorithm to produce an implementation is that all trigger signals be stable [i.e., $EC(u*,k) \subseteq TC(u*,k)$].

The excitation cubes and trigger cubes are easily found with a single pass through the SG. For each excitation region, an excitation cube is built by forming the supercube over all states in the excitation region. The trigger cube is built by finding each signal that takes the circuit into the excitation cube.

Example 6.3.11 The excitation cubes and trigger cubes corresponding to all the excitation regions in the example SG in Figure 6.13(a) are shown in Table 6.1. Notice that every trigger signal is stable and our algorithm proceeds to find the optimal single-cube cover.

The goal of the single-cube algorithm is to find a cube $C(u*,k)$ where $EC(u*,k) \subseteq C(u*,k) \subseteq TC(u*,k)$ such that it satisfies the required correctness constraints for the given technology. The cube should also be maximal (i.e., cover as many states as possible). The single-cube algorithm begins with a cube consisting only of the trigger signals [i.e., $C(u*,k) = TC(u*,k)$]. If this cover contains no states that violate the required correctness constraints,

Table 6.1 Excitation and trigger cubes with cube vector $\langle a, b, c, d \rangle$.

$u*, k$	$EC(u*, k)$	$TC(u*, k)$
$c+, 1$	0100	-1--
$c+, 2$	1101	---1
$c-, 1$	0010	-0--
$d+, 1$	1100	-1--
$d-, 1$	1111	--1-

we are done. This, however, is often not the case, and context signals must be added to the cube to remove any *violating states*. For each violation detected, the procedure determines the choices of context signals which would exclude the violating state. Finding the smallest set of context signals to resolve all violations is a covering problem.

If the implementation technology is generalized C-elements, then for a set region a state is a violating state when the trigger cube intersects a set of states where the signal is falling or stable low. Similarly, for a reset region, a state is a violating state when the trigger cube intersects the set of states where it is rising or stable high. More formally, the sets of violating states are defined as follows:

$$V(u+, k) = \{s \in S \mid s \in TC(u+, k) \wedge s \in ES(u-) \cup QS(u-)\}$$
$$V(u-, k) = \{s \in S \mid s \in TC(u-, k) \wedge s \in ES(u+) \cup QS(u+)\}$$

Example 6.3.12 Returning again to our example, the trigger cube for $ER(c+, 1)$ is -1--, which includes the reachable states $01R0$, $0F10$, $F110$, $111F$, $11R1$, and $110R$. The only violating state is $110R$ since it is in $QS(c-)$. For $ER(c+, 2)$, $ER(c-, 1)$, and $ER(d-, 1)$, there are no violating states. The trigger cube for $ER(d+, 1)$ is -1--, which illegally intersects the states $111F$, $F110$, $0F10$, and $01R0$.

The next thing we need to do is determine which context signals remove these violating states. A signal is allowed to be a context signal if it is stable in the excitation cube [i.e., $EC(u*, k)(v) = 0$ or $EC(u*, k)(v) = 1$]. A context signal removes a violating state when it has a different value in the excitation cube and the violating state. In other words, a context signal v removes a violating state s when $EC(u*, k)(v) = \overline{s(v)}$.

Example 6.3.13 Consider first the violating state $110R$ for $ER(c+, 1)$. The only possible context signal is a, which is 0 in the excitation cube and 1 in the violating state. Therefore, we must reduce the cover to 01-- to remove this violating state. For $ER(d+, 1)$, the violating state $111F$ can be removed using either context signals c or d, $F110$ can be removed with c, $0F10$ can be removed with a or c, and finally, $01R0$ can be removed with a.

In order to select the minimum number of context signals, to remove all the violating states, we need to set up a covering problem. The constraint matrix for the covering problem has a row for each violating state and a column for each context signal.

Example 6.3.14 The constraint matrix for $ER(d+,1)$ is shown below.

	a	c	d
$111F$	–	1	1
$F110$	–	1	–
$0F10$	1	1	–
$01R0$	1	–	–

The context signals a and c are essential and solve the entire matrix. This means that the implementation for the set function for d is $ab\bar{c}$. The final circuit implementation is shown in Figure 6.13(b). Again, the combinational optimization can be applied to obtain the circuit shown in Figure 6.13(c) for signal d.

To produce a standard C-implementation, we must use the covering and entrance constraints. First, for each excitation cube, $EC(u*,k)$, the procedure finds all states in the initial cover [i.e., $TC(u*,k)$] which violate the covering constraint. In other words, a state s in $TC(u*,k)$ is a violating state if the signal u is excited in the opposite direction, is stable at the opposite value, or is excited in the same direction but the state is not in the current excitation region. The set of covering violations, $CV(u*,k)$, can be defined more formally as follows:

$$\begin{aligned} CV(u+,k) &= \{s \in S \mid s \in TC(u+,k) \wedge s \in ES(u-) \cup QS(u-) \\ &\quad \cup (ES(u+) - EC(u+,k))\} \\ CV(u-,k) &= \{s \in S \mid s \in TC(u-,k) \wedge s \in ES(u+) \cup QS(u+) \\ &\quad \cup (ES(u-) - EC(u-,k))\} \end{aligned}$$

Example 6.3.15 With this modification, for $EC(c+,1)$, in addition to the state $110R$, the state $11R1$ is also a violating state since it is in the other set region. The sets of violating states for $EC(c+,2)$, $EC(c-,1)$, and $EC(d-,1)$ are still empty, and the set of violating states for $EC(d+,1)$ is unchanged.

Next, all state transitions which either violate or may violate the entrance constraint must be found. For each state transition $(s_i, v*, s_j)$, this is possible when s_j is a quiescent state, s_j is in the initial cover, and v excludes s_i. The set of entrance violations, $EV(u*,k)$, can be defined more formally as follows:

$$\begin{aligned} EV(u+,k) &= \{s_j \in S \mid (s_i, v*, s_j) \in \delta \wedge s_j \in QS(u+) \wedge s_j \in TC(u+,k) \\ &\quad \wedge EC(u+,k)(v) = \overline{s_i(v)} \\ EV(u-,k) &= \{s_j \in S \mid (s_i, v*, s_j) \in \delta \wedge s_j \in QS(u-) \wedge s_j \in TC(u-,k) \\ &\quad \wedge EC(u-,k)(v) = \overline{s_i(v)} \end{aligned}$$

When a potential entrance violation is detected, a context signal must be added which excludes s_j from the cover when v is included in the cover. Therefore, if v is a trigger signal, the state s_j is a violating state. If v is a possible context signal choice, s_j becomes a violating state when v is included in the cover.

Example 6.3.16 Let's consider $EC(c+,1)$ again. The state transition $(F110,\ a-,\ 0F10)$ is a potential entrance violation since $0F10$ is in $QS(c+)$, $0F10$ is in $TC(c+,1)$, and the signal a excludes the state $F110$ from the cover. Therefore, the state $0F10$ is a violating state when a is included in the cover. Similarly, the state transition $(111F, d-, F110)$ becomes an entrance violation when d is included as a context signal. For $EC(d-,1)$, the state transition $(01R0, c+, 0F10)$ is a potential entrance violation, and since c is a trigger signal, $0F10$ is a violating state.

Again, to select the minimum number of context signals, we have a covering problem. Since inclusion of certain context signals causes some states to have entrance violations, the covering problem is binate. To solve this binate covering problem, we create a constraint matrix for each region. There is a row in the constraint matrix for each violation and each violation that could potentially arise from a context signal choice, and there is a column for each context signal. The entry in the matrix contains a 1 if the context signal excludes the violating state. An entry in the matrix contains a 0 if the inclusion of the context signal would require a new violation to be resolved.

Example 6.3.17 The constraint matrix for $EC(c+,1)$ is shown below.

	a	c	d
$110R$	1	–	–
$11R1$	1	–	1
$0F10$	0	1	–
$F110$	1	1	0

The context signal a is essential, so it must be in the cover. Selecting a, however, causes the state $0F10$ to become a violating state. To exclude this state, the signal c must be added as an additional context signal. Therefore, the implementation of $EC(c+,1)$ is $\overline{a}b\overline{c}$. The standard C-implementation is shown in Figure 6.16(b).

If a violation is detected for which there is no context signal to resolve it, the constraint matrix construction fails. In this case, or if a trigger signal is not stable, we must either constrain concurrency, add state variables, or use the more general algorithm described earlier to find a circuit. We conclude this section with a few examples to show how these situations appear.

Example 6.3.18 First, consider $ER(z+,1)$ for the nondistributive SG shown in Figure 6.19. The excitation cube for this region would be `--0` and the trigger cube would be `11-`. Clearly, the trigger signals are not stable in the excitation region, so no single-cube cover exists.

In general, the single-cube algorithm cannot be applied to nondistributive state graphs.

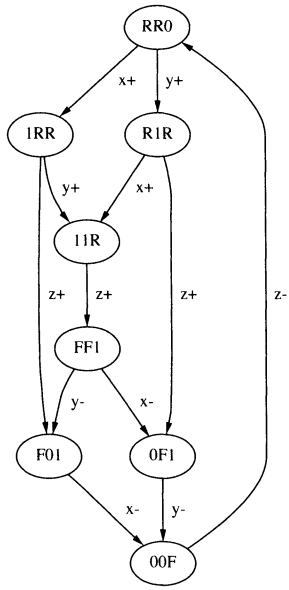

Fig. 6.19 Nondistributive SG [state vector $\langle x, y, z \rangle$].

Example 6.3.19 As a second example, consider $ER(w-, 1)$ for the SG in Figure 6.20(a). The excitation cube is `10--` and the trigger cube is `--0-`. Again, we find that the trigger signal is not stable in the excitation region, so no single-cube cover exists. If we remove the offending state $F010$, we obtain the SG in Figure 6.20(b). The signal w can now be implemented with a single cube, but there is a problem with x. Consider $ER(x+, 1)$ which has excitation cube `00--` and trigger cube `0---`. There is no problem with the trigger signal. The state $R011$ is a violating state since it is included in the trigger cube and x is stable low. There is, however, no possible context signal which can be added to remove this violating state. Therefore, again there is no single-cube cover.

6.4 HAZARD-FREE DECOMPOSITION

The synthesis method described in Section 6.3.4 puts no restrictions on the size of the gates needed to implement the region functions. In all technologies, however, there is some limitation on the number of inputs that a gate can have. For example, in CMOS, it is typically not prudent to put more than four transistors in series, as it significantly degrades performance. Furthermore, large transistor stacks can have charge-sharing problems. These problems are especially dangerous in generalized C-elements, where excess charge can lead to the gate latching an incorrect value. Therefore, it is often necessary to

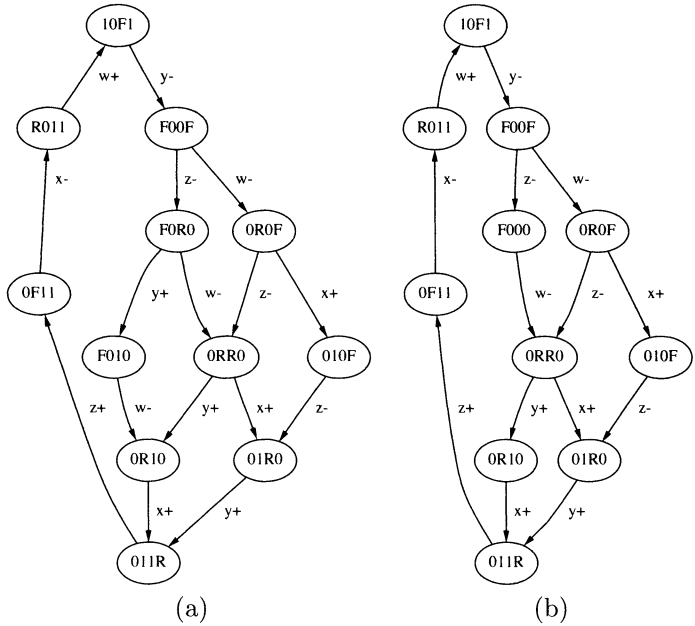

Fig. 6.20 (a) SG with an unstable trigger signal (state vector $\langle w, x, y, z \rangle$). (b) SG with an unresolvable violation (state vector $\langle w, x, y, z \rangle$).

decompose high-fanin gates into limited-fanin gates found in the given gate library. For Huffman circuits, decomposition of high-fanin gates can be done in an arbitrary fashion, preserving hazard freedom (if it existed in the original circuit). Unfortunately, this problem is much more difficult for Muller circuits.

Example 6.4.1 Let us assume that we have a library of gates which allows no more than two inputs per gate. Therefore, for the circuit shown in Figure 6.13(c), we would need to decompose the three-input AND gate into two two-input AND gates. Two possible ways of decomposing this gate are shown in Figure 6.21(b) and (c). Consider first the circuit shown in Figure 6.21(b). In state $\langle F110 \rangle$, inputs a, b, c, and internal signal e are high while d is low. After a falls, we move to state $\langle 0F10 \rangle$, and e becomes excited to go low. However, let us assume that the AND-gate generating signal e is slow. Next, b falls, moving us to state $\langle 00F0 \rangle$. If at this point c falls before e falls, d can become excited to rise prematurely. The result is that there is a hazard on the signal d, and there is a potential for a circuit failure.

Next, consider the circuit shown in Figure 6.21(c), beginning in state $\langle F110 \rangle$. This time e begins low, and it does not become excited to change until after a falls, b falls, c falls, and a rises again. At this point, however, b is already low, which maintains d in its low state until b rises again. In fact, there is no sequence of transitions that can cause this circuit to experience a hazard.

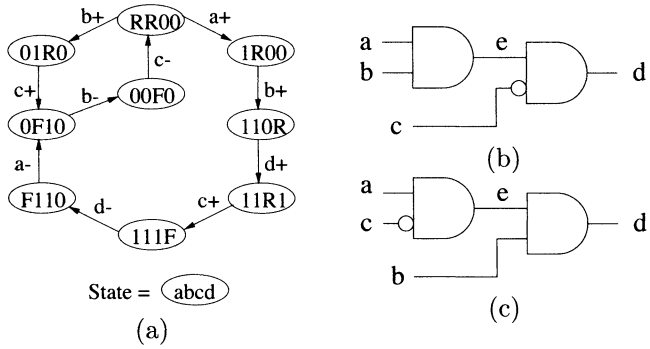

Fig. 6.21 (a) SG for example. (b) Hazardous and (c) hazard-free decomposition.

This example illustrates the need for special care during decomposition to guarantee a hazard-free implementation. Essentially, we need to find a new internal signal which can be added to produce a simpler circuit. Therefore, in this section we present a simple technique for finding hazard-free decompositions which is similar to the insertion point procedure described earlier to solve CSC violations.

6.4.1 Insertion Points Revisited

Let us analyze the circuits in Figure 6.21 using the idea of insertion points. For the hazard-free circuit shown in Figure 6.21(c), the transition point to set the new signal e is $(\{a+\};\{d+\})$. This means that e is rising in the states $\langle 1000\rangle$ and $\langle 110R\rangle$, but it is guaranteed to have risen before d can rise. In other words, d rising acknowledges that e has risen. The transition point to reset the new signal is $(\{c+\},\{d-\})$. This means that e is falling only in state $\langle 111F\rangle$, and it is guaranteed to have fallen before d can fall. Therefore, again d changing acknowledges the change on e. The fact that changes on this internal signal are acknowledged by primary output changes is what guarantees hazard freedom.

Now consider the hazardous circuit shown in Figure 6.21(b). The transition point to set e is $(\{b+\},\{d+\})$, so it is changing only in state $\langle 110R\rangle$. This means that again d rising acknowledges that e has risen. The transition point for e to fall has a start set of $\{a-\}$, but it has no end set. There is no transition that is prevented from occurring until e has fallen. It is this lack of acknowledgment that leads to the hazard described previously. One could consider using $b-$ in an end set, but this is an input signal, and we are not allowed to change the interface behavior. If, instead, we use $c-$ as an end set, we just move the problem, as e now requires a three-input gate.

Again, we have a large number of potential insertion points which need to be filtered. The filters discussed for CSC violations still apply. Due to

the nature of the decomposition problem, however, there are some additional restrictions that can be used.

Consider the decomposition of a cover, $C(u*, k)$, which is composed of a single cube, or AND gate. For the new signal to be useful to decompose this gate, either its set or reset function must be composed of signals in the original AND gate. Therefore, we can restrict the start set for one of the transition points to transitions on just those signals in the gate being decomposed. We also know that this original gate is composed of trigger and context signals. By their definition, context signals always change before trigger signals, so they are not concurrent with the trigger signals. Therefore, we only need to consider start sets which include either trigger or context signals from the original gate, but not both. We need to consider all possible combinations of the trigger signals as potential start sets, but we only need to consider concurrent subsets of the context signals as potential start sets. Finally, since this new signal transition must complete before $u*$ can be excited, we only need to consider transitions that occur after those in the start set and before $u*$ as potential candidates to be in the end set.

If both the cover of a set region and a reset region of u must be decomposed, the same restrictions on a transition point can be used for the reverse transition on the new signal. If, however, the new signal is not needed to decompose both a set and reset region, there is a little more flexibility in the start and end sets. First, the start set should include concurrent transitions which occur after $u*$ and before any transitions in the first start set. Including the reverse transition of $u*$ in the end set is often useful, but any transition after $u*$ could potentially be used in the end set.

6.4.2 Algorithm for Hazard-Free Decomposition

The complete algorithm for hazard-free decomposition is shown in Figure 6.22. This algorithm takes a SG and an initial design. It then finds all gates which have a fanin larger than the maximum allowed size. If there are none, the algorithm returns the current design. Otherwise, it records the current design as the best found so far, and it finds all transition points which can be used to decompose the high-fanin gates. It then considers each insertion point in turn. If the state graph colored using a legal insertion point is consistent, it inserts the new signal and resynthesizes the design. If the resulting design has less high-fanin gates or has a lower cost, it is recorded. Again, a single state signal may not decompose all high-fanin gates, so this algorithm recursively calls itself to add more signals until all high-fanin gates have been decomposed. This algorithm is illustrated using the following example.

> **Example 6.4.2** Consider the SG shown in Figure 6.8 and its circuit implementation shown in Figure 6.12(a). Let us assume that we are only allowed gates with at most three inputs. Therefore, we need to decompose the four-input AND gate used to implement the set region for *ack_wine*. For this cover, the trigger signals are *req_wine* and *ack_patron*,

```
decomposition(SG, design, maxsize) {
  HF = find_high_fanin_gates(design, maxsize);
  if (|HF| = 0) return design;        /* No high-fanin gates, return.*/
  best = |HF|;
  best_IP = design;                          /* Initialize best found.*/
  TP = find_all_transition_points(SG, design, HF);
  foreach TP_R ∈ TP
    foreach TP_F ∈ TP
      if IP = (TP_R, TP_F) is legal then {
        CSG = color_state_graph(SG, TP_R, TP_F);
        if (CSG is consistent) then {
          SG' = insert_state_signal(SG, IP);
          design = synthesis(SG');              /* Find new circuit.*/
          HF = find_high_fanin_gates(design, maxsize);
          if ((|HF| < best) or ((|HF| = best) and
            (cost(design) < cost(best_IP)))) then {
            best = |HF|;
            best_IP = design;
        }
      }
  design = decomposition(SG, design);          /* Add more signals.*/
  return design;
}
```

Fig. 6.22 Algorithm for decomposing high-fanin gates.

and the context signals are *req_patron* and *CSC0*. We would like the new gate to be off the critical path, so we first attempt to implement it using context signals. The transition *CSC0*− always precedes *req_patron*−, so these two transitions cannot be included in the same start set. Therefore, there are only two possible transition points using context signals:

$$(\{CSC0-\}, \{ack\_wine+\})$$
$$(\{req\_patron-\}, \{ack\_wine+\})$$

If we need to use the trigger signals instead, there are three possible transition points to consider:

$$(\{req\_wine+\}, \{ack\_wine+\})$$
$$(\{ack\_patron-\}, \{ack\_wine+\})$$
$$(\{req\_wine+, ack\_patron-\}, \{ack\_wine+\})$$

Next, we need to consider the transition points for the reverse transition of the new signal. We would like to insert the reverse transition somewhere between *ack_wine*+ and *ack_wine*−, so we restrict our attention to transitions between them. In other words, the start and end sets are only made up of the following transitions: *ack_wine*+, *CSC0*+, *req_wine*−, and *ack_wine*−. Consider first using *ack_wine*+ in the start

set. If we do this, no other transition can be in the start set, since all of the other choices occur after *ack_wine+*. Of the remaining transitions, the end set cannot include *req_wine−*, since it is an input transition. Therefore, there are two possible transition points:

$$(\{ack\_wine+\}, \{CSC0+\})$$
$$(\{ack\_wine+\}, \{ack\_wine-\})$$

If we use *CSC0+* in the start set, its end set must be *ack_wine−*, since *req_wine* is an input. If we use *req_wine−* in the start set, either *CSC0+* or *ack_wine−* can be in the end set. Finally, since *CSC0+* and *req_wine−* change concurrently, they can appear in the start set together. In this case, the end set must be *ack_wine−*. Therefore, we have the following transition points:

$$(\{CSC0+\}, \{ack\_wine-\})$$
$$(\{req\_wine-\}, \{CSC0+\})$$
$$(\{req\_wine-\}, \{ack\_wine-\})$$
$$(\{CSC0+, req\_wine-\}, \{ack\_wine-\})$$

Now that we have enumerated all the transition points, the next step is to form insertion points out of combinations. We again color the graph to determine if the insertion point leads to a consistent state assignment. We should also check if any USC violations become CSC violations as a result of the signal insertion. If neither of these problems arise, we need to further check the insertion point by deriving a new state graph and synthesizing the circuit. If the new circuit meets the fanin constraints, the insertion point is accepted. If not, we try the next insertion point.

For our example, we must select the following transition point for one transition of the new signal:

$$(\{req\_patron-\}, \{ack\_wine+\})$$

However, we are free to select any of the reverse transition points to meet the gate size of 3 constraint. An example circuit is shown in Figure 6.23 using the following transition point for the reverse transition:

$$(\{ack\_wine+\}, \{ack\_wine-\})$$

6.5 LIMITATIONS OF SPEED-INDEPENDENT DESIGN

The circuit shown in Figure 6.23 requires several gates with inverted inputs. If the bubble on the *ack_patron* input to the set AND gate for *ack_wine* is removed and replaced with an inverter, the circuit is no longer hazard-free. Consider the state of the circuit just after *ack_wine* has gone low and before

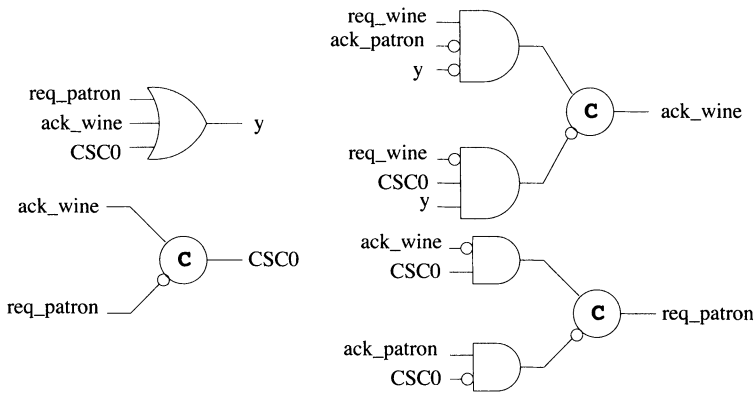

Fig. 6.23 Passive/active wine shop using gates with a maximum stack size of 3.

req_patron has gone high. In this state *CSC0* is high, so *req_patron* can go high followed by *ack_patron* going high. This enables this new inverter to go low. In the meantime, *CSC0* can go low, followed by *req_patron* going low, and finally, *ack_patron* can go low, resulting in the inverter being disabled. In other words, this sequence would result in a hazard at the output of this new inverter. Clearly, this long sequence of events is highly unlikely to be faster than the switching time of an inverter. However, under the speed-independent delay model, this sequence must be considered to be possible. If timing information is known, however, it may be possible to determine that such a hazard is impossible. This is the subject of the next chapter.

6.6 SOURCES

Speed-independent switching circuit theory originated with Muller and Bartky [277, 278, 279]. Perhaps, the best description of Muller's work is in Volume II of Miller's textbook on switching theory [272]. The definition of speed independence given in Section 6.1 follows that given in [272, 279]. Several of the concepts and examples in Section 6.1 came from this early work [272, 279]. Many of the definitions, however, have been tuned to fit the speed-independent design methods described in the following sections. A method for efficient identification of speed-independent circuits is given by Kishinevsky et al. [202].

Similar to the speed-independent model is the *self-timed* model proposed by Seitz [345, 347]. In this model, wires in *equipotential regions* are assumed to have zero delay. The length of these wires must be short enough that a change on one end can be perceived at the other end in less than the *transit time* of an electron through a transistor. If delay elements are added to all wires which

are not in an equipotential regions, this model is reduced to essentially the speed-independent model.

The complete state coding problem and a method to solve it was first described by Bartky [24]. The method for solving the CSC problem described in Section 6.2 follows the work of Krieger [211]. The idea of coloring the state graph originated with Vanbekbergen et al. [391]. This work often produced inefficient circuits since it allowed the state graph to be colored arbitrarily to solve the CSC problem. To improve upon the logic generated, Ykman-Couvreur and Lin restricted the coloring such that the new state signals are only inserted into excitation and switching regions [412, 413]. The insertion points described here are a generalization of this idea. Cortadella et al. generalized the idea further, expanding the set of possible insertion points [93]. A quite different approach taken to solving the CSC problem is taken by Lavagno et al. [223]. In this work, the state graph is converted to an FSM, and traditional FSM critical race free state assignment approach is taken. Another approach proposed by Gu and Puri first decomposes the STG specification of the circuit into manageable piece, and it then applies Boolean satisfiability to find a solution for each of the smaller subgraphs [157].

Most early speed-independent design approaches required complex atomic gates as well as several of the more recent design methodologies proposed by Chu [83], Meng et al. [268], and Vanbekbergen [389, 390]. A speed-independent design method which targets three-valued logic can be found in [405]. Martin's speed-independent design methodology was the first to utilize generalized C-elements [249, 254]. The first work to synthesize speed-independent circuits using only basic gates (NANDs and NORs) was done by Varshavsky and his students [393]. This work, however, produced rather inefficient circuits and was limited to circuits without choice. Adding the C-element to the list of basic gates, Beerel and Meng developed conditions for correct standard C-implementations [25]. Similar conditions with some generalizations were later presented by Kondratyev et al. [210]. Quite a different approach is taken by Sawasaki et al. which uses hazardous set and reset logic but uses a special flip-flop rather than a C-element to filter the hazards, keeping the outputs hazard-free [339]. The single-cube algorithm described in Section 6.3.4 was developed by Myers [283]. A comparison of this algorithm to the more general algorithm appears in [29]. The theory in Section 6.3.3 comes from this paper. A method that uses BDDs to find all possible correct covers is presented in [377]. A synthesis technique that assigned don't cares in such a way as to ensure initializability is given in [73]. Several researchers have developed methods to synthesize gate-level hazard-free speed-independent circuits directly from a STG [185, 187, 235, 305, 350]. This avoids the state explosion inherent in SG-based synthesis methods.

The decomposition problem was first discussed by Kimura, who called these decompositions, *extensions* [197, 198]. An extension is called a *good extension* if its behavior ignoring the new signals produces exactly the same allowed sequences as the original. In particular, Kimura investigated the effect of adding

buffers into a wire and stated that a circuit suffers the *delay problem of the first kind* when adding buffers to wires that have delay results in a bad extension [197]. In [198] he introduced delay problems of the second kind, in which the circuit is no longer partially semi-modular with respect to the original signals, and the third kind, in which the circuit is not totally semi-modular. The work by Varshavsky and his students developed methods to decompose logic, but as mentioned earlier, they were limited in their utility and often produced inefficient circuits [393]. Lavagno et al. leveraged synchronous technology mapping methods by producing a circuit that is hazard-free using the atomic gate assumption, but then added delay elements to remove hazards introduced by decomposition [221, 222, 225]. Siegel and De Micheli developed conditions in which high-fanin AND gates in a standard C-implementation can be decomposed [351, 353]. Myers et al. introduced an approach to breaking up high-fanin gC gates using a method of decomposition and resynthesis [284]. Perhaps one of the most important works in this area is Burns's method which utilizes implicit methods to explore large families of potential decompositions of a generalized C-element implementation [66]. Recent work by Cortadella et al. and Kondratyev et al. has built upon this approach to allow for a greater range of potential decompositions [92, 209]. The discussion in Section 6.4 was inspired by Krieger's insertion point idea [211] and recent work by Burns and others.

Problems

6.1 Speed Independence

For the SG shown in Figure 6.24(a), find its corresponding partial order of equivalence classes. A property is said to hold in a state if it holds in all states that follow it. For each state in the SG, determine which of the following properties hold:

1. Speed independence
2. Semi-modularity
3. Distributivity
4. Totally sequential

6.2 Complete State Coding

For the STG in Figure 6.24(b), find the state graph and then find all state pairs which violate complete state coding. Solve the CSC violations by adding a new state variable. Show the new state graph.

6.3 Hazard-Free Logic Synthesis

From the SG in Figure 6.25, find a hazard-free logic implementation for the output signals x, d, and c using the:

6.3.1. Atomic gate approach.

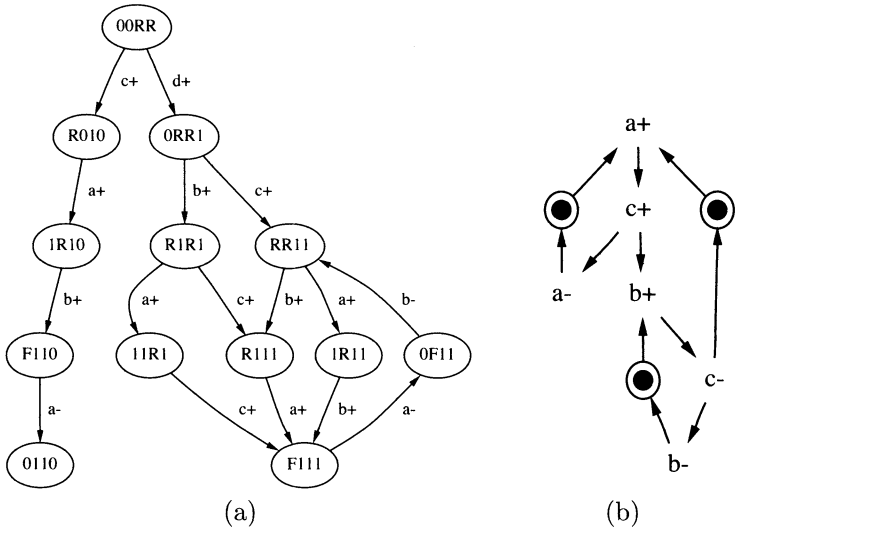

Fig. 6.24 (a) SG for Problem 6.1. (b) STG for Problem 6.2.

6.3.2. Generalized C-element approach (use the single-cube algorithm).

6.3.3. Standard C-element approach (use the single-cube algorithm).

6.4 Exceptions

From the SG shown in Figure 6.26 for the output signals $r2$, $a1$, and x:

6.4.1. Find the excitation cubes and trigger cubes.

6.4.2. Use the single-cube algorithm to find a standard C-implementation, if possible. If not possible, explain why not.

6.4.3. Use the more general algorithm to find a multicube cover for the unimplemented signals from 6.4.2.

6.5 Speed-Independent Design

Perform the following on the VHDL in Figure 6.27.

6.5.1. Assuming that all signals are initially low, find the state graph.

6.5.2. Find all state pairs which violate complete state coding.

6.5.3. Solve the CSC violations by using reshuffling. Show the new state graph.

6.5.4. Solve the CSC violations by adding a new state variable q. Show the new state graph.

6.5.5. Use Boolean minimization to find the circuit from either of the solutions above.

6.5.6. Find the circuit from your reshuffled and state variable solutions using a generalized C-element implementation technique. Comment on which is best and why.

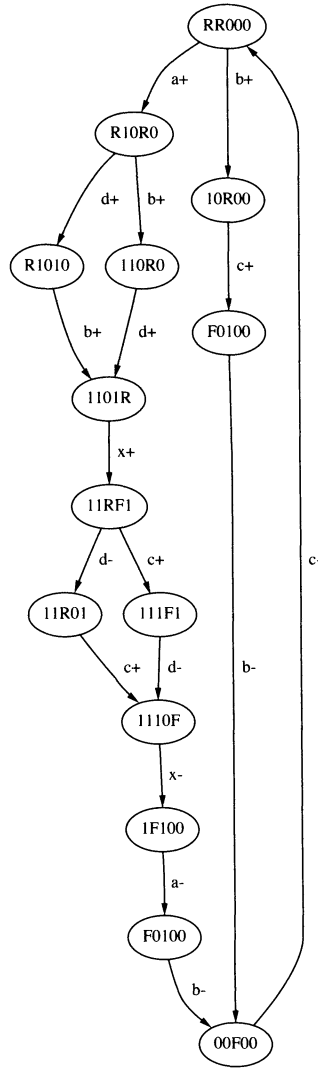

Fig. 6.25 SG for Problem 6.3.

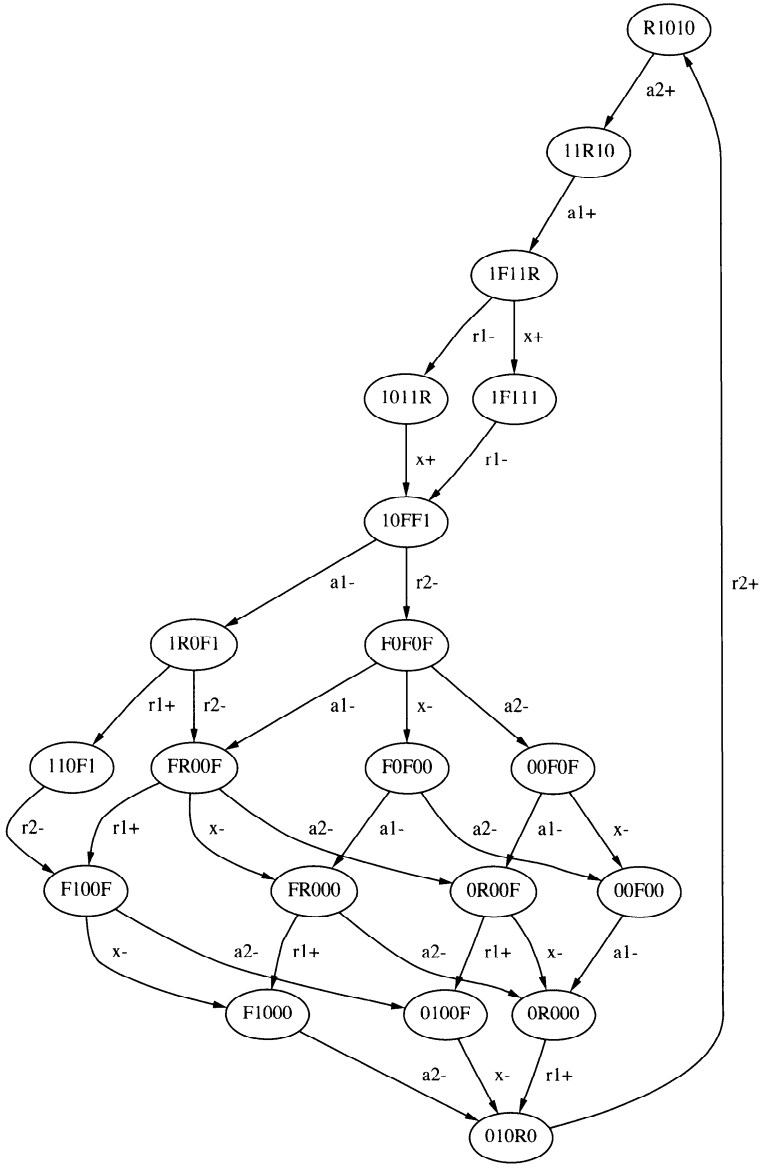

Fig. 6.26 SG for Problem 6.4.

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;
entity p6 is
end entity;
architecture hse of p6 is
  signal ai,bi,x:std_logic; --@ in
  signal ao,bo:std_logic;
begin
main:process
begin
  guard(ai,'1');
  if (x = '1') then
    assign(ao,'1',1,2);
    guard(ai,'0');
    assign(ao,'0',1,2);
  else
    assign(bo,'1',1,2);
    guard(bi,'1');
    assign(bo,'0',1,2);
    guard(bi,'0');
    assign(ao,'1',1,2);
    guard(ai,'0');
    assign(ao,'0',1,2);
  end if;
end process;
ai:process
  variable z:integer;
begin
  z:=selection(2);
  if (z=1) then
    assign(x,'1',1,2);
    assign(ai,'1',1,2);
    guard(ao,'1');
    assign(ai,'0',1,2);
    assign(x,'0',1,2);
    guard(ao,'0');
  else
    assign(ai,'1',1,2);
    guard(ao,'1');
    assign(ai,'0',1,2);
    guard(ao,'0');
  end if;
end process;  bi:process
begin
  guard(bo,'1');
  assign(bi,'1',1,2);
  guard(bo,'0');
  assign(bi,'0',1,2);
end process;
end hse;
```

Fig. 6.27 VHDL for Problem 6.5.

6.6 Standard C-Implementation

Find a standard C-implementation for the circuit specified in Figure 6.28.

6.7 Hazard-Free Decomposition

For the state graph shown in Figure 6.29 and output signals *lo*, *ro*, and *x*:

6.7.1. Find a gC implementation using the single-cube algorithm.

6.7.2. Use the insertion point method to decompose any gates which have a fanin greater than two.

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;
entity p6 is
end entity;
architecture hse of p6 is
  signal ai:std_logic; --@ in
  signal bi:std_logic; --@ in
  signal x:std_logic; --@ in
  signal ao:std_logic;
  signal bo:std_logic;
begin
main:process
begin
  guard(ai,'1');
  if (x = '1') then
    assign(ao,'1',1,2);
    guard(ai,'0');
    assign(ao,'0',1,2);
  else
    assign(bo,'1',1,2);
    guard(bi,'1');
    assign(ao,'1',1,2);
    assign(bo,'0',1,2);
    guard_and(bi,'0',ai,'0');
    assign(ao,'0',1,2);
  end if;
end process;
ai:process
  variable z:integer;
begin
  z:=selection(2);
  if (z=1) then
    assign(x,'1',1,2);
    assign(ai,'1',1,2);
    guard(ao,'1');
    assign(x,'0',1,2);
    assign(ai,'0',1,2);
    guard(ao,'0');
  else
    assign(ai,'1',1,2);
    guard(ao,'1');
    assign(ai,'0',1,2);
    guard(ao,'0');
  end if;
end process;  bi:process
begin
  guard(bo,'1');
  assign(bi,'1',1,2);
  guard(bo,'0');
  assign(bi,'0',1,2);
end process;
end hse;
```

Fig. 6.28 VHDL for Problem 6.6.

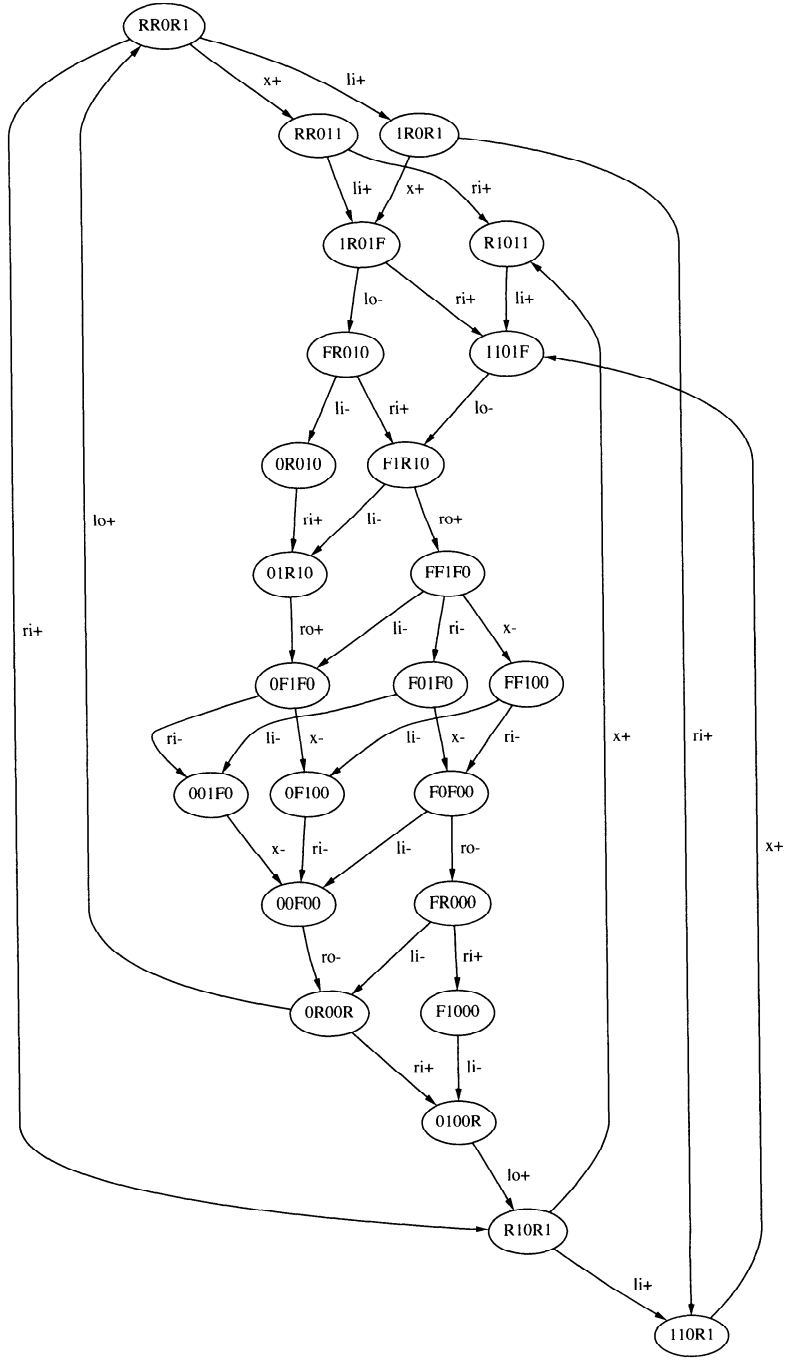

Fig. 6.29 SG for Problem 6.7.