that may be faster and/or better suited for implementation with available logic.

- *Describing finite state machine in HDLs.* You will learn how to describe finite state machines and their structure using hardware description languages.

## 8.1 State Minimization/Reduction

State reduction identifies and combines states that have *equivalent behavior*. Two states have equivalent behavior if, for all input combinations, their outputs are the same and they change to the same or equivalent next states.

For example, in Figure 7.50(b), states $S_0$ and $S_2$ are equivalent. Both states output a 0; both change to $S_1$ on a 1 and self-loop on a 0. Combining these into a single state leads to Figure 7.50(a). On all input strings, the output sequence of either state diagram is exactly the same.

Algorithms for state reduction begin with the symbolic state transition table. First, we group together states that have the same state outputs (Moore machine) or transition outputs (Mealy machine). These are potentially equivalent, since states cannot be equivalent if their outputs differ.

Next, we examine the transitions to see if they go to the same next state for every input combination. If they do, the states are equivalent and we can combine them into a renamed new state. We then change all transitions into the newly combined states and repeat the process until no additional states can be combined.

In the following two subsections, we examine alternative algorithms for state reduction: *row matching* and *implication charts*. The former is a good pencil-and-paper method, but does not always obtain the best reduced state table. Implication charts are more complex to use by hand, but they are easy to implement in software and do find the best possible solution.

We can always combine the two approaches. Row matching quickly reduces the number of states. The implication chart method, now working with fewer states, finds the equivalent states missed by row matching more rapidly.

### 8.1.1 Row-Matching Method

Let's begin our investigation of the row-matching method with a detailed example. We will see how to transform an initial state diagram for a simple sequence detector into a minimized, equivalent state diagram.

***Four-Bit Sequence Detector*** We'll start with an example finite state machine that, if implemented straighforwardly, generates many equivalent states. A common sequential circuit is a sequence detector that looks for a particular pattern of bits on a single input.

*Specification and Initial State Diagram* Let's consider a sequence-detecting finite state machine with the following specification. The machine has a single input $X$ and output $Z$. The output is asserted after each 4-bit input sequence if it consists of one of the binary strings 0110 or 1010. The machine returns to the reset state after each and every 4-bit sequence.

We will assume a Mealy implementation. Some sample behavior of the finite state machine is

$$X = 0010\ 0110\ 1100\ 1010\ 0011\ldots$$

$$Z = 0000\ 0001\ 0000\ 0001\ 0000\ldots$$

The output is asserted only after the previous four serial inputs match one of the specified strings. Also, the input patterns do not overlap: the machine makes a decision to assert its output after each group of 4 bits.

Because this finite state machine recognizes finite length strings, we can place an upper bound on the number of states needed to recognize any particular binary string of length four. Figure 8.1 shows the Mealy state diagram. There are 16 unique paths through the state diagram, one for each possible 4-bit pattern. This adds up to 15 states and 30 transitions. We highlight the paths leading to recognition of the strings 0110 and 1010 in the figure. Only two of the transitions have a 1 output, representing the accepted strings.

*State Table and Row-Matching Method* We can combine many of the states in Figure 8.1 without changing the input/output behavior of the finite state machine. For example, it is clear that once we have the input 00 that there will be no way that we will detect one of our two target strings. But how do we find these equivalent states in a systematic fashion?

First, we look at the state transition table, as shown in Figure 8.2. This table is in a slightly different format than we have seen so far. It contains one row per state, with multiple next-state and output columns

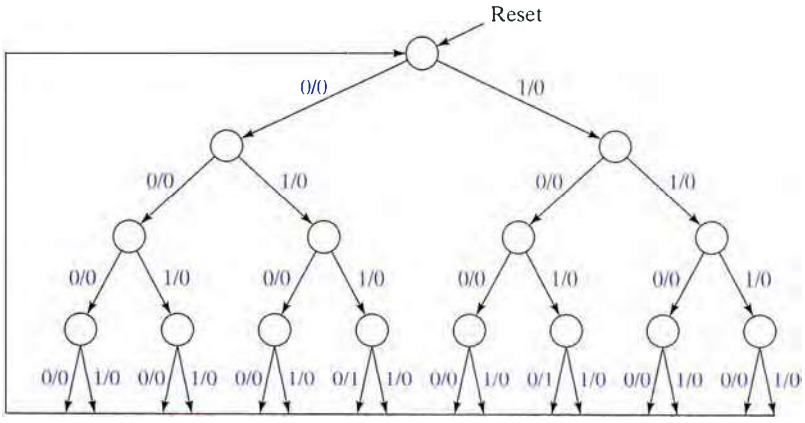


**Figure 8.1** Original state diagram for 4-bit string recognizer.

| Input Sequence | Present State | Next State $X=0$ | Next State $X=1$ | Output $X=0$ | Output $X=1$ |
|---|---|---|---|---|---|
| Reset | $S_0$ | $S_1$ | $S_2$ | 0 | 0 |
| 0 | $S_1$ | $S_3$ | $S_4$ | 0 | 0 |
| 1 | $S_2$ | $S_5$ | $S_6$ | 0 | 0 |
| 00 | $S_3$ | $S_7$ | $S_8$ | 0 | 0 |
| 01 | $S_4$ | $S_9$ | $S_{10}$ | 0 | 0 |
| 10 | $S_5$ | $S_{11}$ | $S_{12}$ | 0 | 0 |
| 11 | $S_6$ | $S_{13}$ | $S_{14}$ | 0 | 0 |
| 000 | $S_7$ | $S_0$ | $S_0$ | 0 | 0 |
| 001 | $S_8$ | $S_0$ | $S_0$ | 0 | 0 |
| 010 | $S_9$ | $S_0$ | $S_0$ | 0 | 0 |
| 011 | $S_{10}$ | $S_0$ | $S_0$ | 1 | 0 |
| 100 | $S_{11}$ | $S_0$ | $S_0$ | 0 | 0 |
| 101 | $S_{12}$ | $S_0$ | $S_0$ | 1 | 0 |
| 110 | $S_{13}$ | $S_0$ | $S_0$ | 0 | 0 |
| 111 | $S_{14}$ | $S_0$ | $S_0$ | 0 | 0 |

**Figure 8.2** Initial state transition table for the 0110 or 1010 sequence detector.

| Input Sequence | Present State | Next State $X=0$ | Next State $X=1$ | Output $X=0$ | Output $X=1$ |
|---|---|---|---|---|---|
| Reset | $S_0$ | $S_1$ | $S_2$ | 0 | 0 |
| 0 | $S_1$ | $S_3$ | $S_4$ | 0 | 0 |
| 1 | $S_2$ | $S_5$ | $S_6$ | 0 | 0 |
| 00 | $S_3$ | $S_7$ | $S_8$ | 0 | 0 |
| 01 | $S_4$ | $S_9$ | $S'_{10}$ | 0 | 0 |
| 10 | $S_5$ | $S_{11}$ | $S'_{10}$ | 0 | 0 |
| 11 | $S_6$ | $S_{13}$ | $S_{14}$ | 0 | 0 |
| 000 | $S_7$ | $S_0$ | $S_0$ | 0 | 0 |
| 001 | $S_8$ | $S_0$ | $S_0$ | 0 | 0 |
| 010 | $S_9$ | $S_0$ | $S_0$ | 0 | 0 |
| 011 or 101 | $S'_{10}$ | $S_0$ | $S_0$ | 1 | 0 |
| 100 | $S_{11}$ | $S_0$ | $S_0$ | 0 | 0 |
| 110 | $S_{13}$ | $S_0$ | $S_0$ | 0 | 0 |
| 111 | $S_{14}$ | $S_0$ | $S_0$ | 0 | 0 |

**Figure 8.3** Revised state transition table after $S_{10}$ and $S_{12}$ are combined.

based on the input combinations. It gives exactly the same information as a table with separate rows for each state and input combination.

The input sequence column is a documentation aid, describing the partial string as seen so far. When read from left to right, it describes the sequence of input bits that lead to the given state.

Next we examine the rows of the state transition table to find any with identical next-state and output values (hence the term "row matching"). For this finite state machine, we can combine $S_{10}$ and $S_{12}$. Let's call the new state $S'_{10}$ and use it to rename all transitions to $S_{10}$ or $S_{12}$. The revised state table is shown in Figure 8.3.

| Input Sequence | Present State | Next State $X=0$ | Next State $X=1$ | Output $X=0$ | Output $X=1$ |
|---|---|---|---|---|---|
| Reset | $S_0$ | $S_1$ | $S_2$ | 0 | 0 |
| 0 | $S_1$ | $S_3$ | $S_4$ | 0 | 0 |
| 1 | $S_2$ | $S_5$ | $S_6$ | 0 | 0 |
| 00 | $S_3$ | $S_7'$ | $S_7'$ | 0 | 0 |
| 01 | $S_4$ | $S_7'$ | $S_{10}'$ | 0 | 0 |
| 10 | $S_5$ | $S_7'$ | $S_{10}'$ | 0 | 0 |
| 11 | $S_6$ | $S_7'$ | $S_7'$ | 0 | 0 |
| Not (011 or 101) | $S_7'$ | $S_0$ | $S_0$ | 0 | 0 |
| 011 or 101 | $S_{10}'$ | $S_0$ | $S_0$ | 1 | 0 |

**Figure 8.4** **Revised state transition table after $S_7$, $S_8$, $S_9$, $S_{11}$, $S_{13}$, and $S_{14}$ are combined.**

| Input Sequence | Present State | Next State $X=0$ | Next State $X=1$ | Output $X=0$ | Output $X=1$ |
|---|---|---|---|---|---|
| Reset | $S_0$ | $S_1$ | $S_2$ | 0 | 0 |
| 0 | $S_1$ | $S_3'$ | $S_4'$ | 0 | 0 |
| 1 | $S_2$ | $S_4'$ | $S_3'$ | 0 | 0 |
| 00 or 11 | $S_3'$ | $S_7'$ | $S_7'$ | 0 | 0 |
| 01 or 10 | $S_4'$ | $S_7'$ | $S_{10}'$ | 0 | 0 |
| Not (011 or 101) | $S_7'$ | $S_0$ | $S_0$ | 0 | 0 |
| 011 or 101 | $S_{10}'$ | $S_0$ | $S_0$ | 1 | 0 |

**Figure 8.5** **Final reduced state transition table after $S_3$, $S_6$ and $S_4$, $S_5$ are combined.**

*Row-Matching Iteration* We continue matching rows until we can no longer combine any. In Figure 8.3, $S_7$, $S_8$, $S_9$, $S_{11}$, $S_{13}$, and $S_{14}$ all have the same next states and outputs. We combine them into a renamed state $S_7'$. The table, with renamed transitions, is shown in Figure 8.4.

Now states $S_3$ and $S_6$ can be combined, as can $S_4$ and $S_5$. We call the combined states $S_3'$ and $S_4'$, respectively. The final reduced state transition table is shown in Figure 8.5. In the process, we have reduced 15 states to just 7 states. This allows us encode the state in 3 bits rather than 4. The reduced state diagram is given in Figure 8.6.

*Limitations of the Row-Matching Method* Unfortunately, row matching does not always yield the most-reduced state table. We can prove this with a simple counterexample. Figure 8.7 shows the state table for the three-state odd parity checker of Figure 7.50. Although states $S_0$ and $S_2$ have the same output, they do not have the same next state. Thus, they cannot be combined by simple row matching. The problem is the self-loop transitions on input 0. If we combined these two states, the self-loop would be maintained, but this is not found by row matching. We need another, more rigorous method for state reduction.

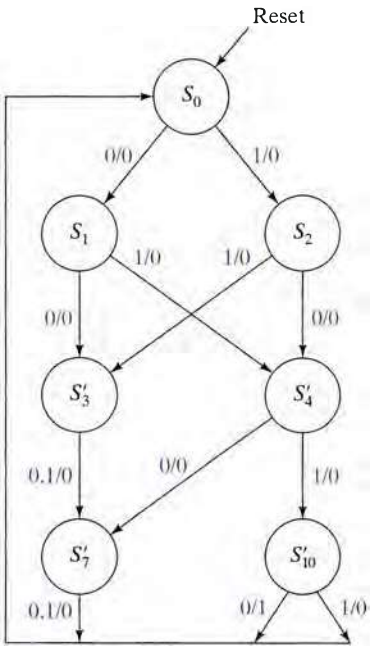


**Figure 8.6** **Reduced state diagram for 4-bit string recognizer.**

| Present State | Next State $X=0$ | Next State $X=1$ | Output |
|---|---|---|---|
| $S_0$ | $S_0$ | $S_1$ | 0 |
| $S_1$ | $S_1$ | $S_2$ | 1 |
| $S_2$ | $S_2$ | $S_1$ | 0 |

**Figure 8.7** **State table for three-state odd parity checker.**

| Input Sequence | Present State | Next State $X=0$ | Next State $X=1$ | Output $X=0$ | Output $X=1$ |
|---|---|---|---|---|---|
| Reset | $S_0$ | $S_1$ | $S_2$ | 0 | 0 |
| 0 | $S_1$ | $S_3$ | $S_4$ | 0 | 0 |
| 1 | $S_2$ | $S_5$ | $S_6$ | 0 | 0 |
| 00 | $S_3$ | $S_0$ | $S_0$ | 0 | 0 |
| 01 | $S_4$ | $S_0$ | $S_0$ | 1 | 0 |
| 10 | $S_5$ | $S_0$ | $S_0$ | 0 | 0 |
| 11 | $S_6$ | $S_0$ | $S_0$ | 1 | 0 |

**Figure 8.8** **Initial state transition table for the 3-bit sequence detector.**

### 8.1.2 Implication Chart Method

The implication chart method is a more systematic approach to finding the states that can be combined into a single reduced state. As you might suspect, the method is more complex and is better suited for machine implementation than hand use.

*Three-Bit Sequence Detector* We'll use a slightly smaller sequence detector that looks at only 3 bits at a time to help us highlight this method. It starts off with 7 states, instead of 15.

*Specification and Initial State Table* Your goal is to design a binary sequence detector that will output a 1 whenever the machine has observed the serial sequence 010 or 110 at the inputs. Figure 8.8 shows its initial state table.

*Data Structure: The Implication Chart* The method operates on a data structure that enumerates all possible combinations of states taken two at a time, called an *implication chart*. Figure 8.9(a) shows the chart with an entry for every pair of states. This form of the chart is more complicated than it needs to be. For example, the diagonal entries are not needed: it does not reduce states to combine a state with itself! And the upper and lower triangles of entries are symmetric. The chart entry for $S_i$ and $S_j$ contains the same information as that for $S_j$ and $S_i$. Thus, we work with the reduced structure of Figure 8.9(b). You can now see that if we had done a 4-bit sequence detector, we would have had a chart with 105 cells instead of the 21 of Figure 8.9. In general, for $n$ starting states, our implication chart data structure will require $(n^2 - n)/2$ cells.

We fill in the implication chart as follows. Let $X_{ij}$ be the entry whose row is labeled by state $S_i$ and whose column is labeled by state $S_j$.