thus reducing circuit size. Reducing the number of states is possible when the FSM contains states that are equivalent to one another. For example, consider the FSM of Figure 6.35(a), having input x and output y. Examination reveals that states *S2* and *S3* appear to be the same as states *S0* and *S1*. Regardless of whether we start in *S0* or *S2*, the outputs will be identical. For example, if we start in *S0* and the input sequence for four clock edges is 1, 1, 0, 0, the state sequence will be *S0*, *S1*, *S1*, *S2*, *S2*, so the output sequence will be 0, 1, 1, 0, 0. If instead we start in *S2*, the same input sequence will result in a state sequence of *S2*, *S3*, *S3*, *S0*, *S0*, so the output sequence will again be 0, 1, 1, 0, 0. In fact, if we tried all possible input sequences, we would find that the output sequence starting from state *S0* would be identical to the output sequence starting from state *S2*. States *S0* and *S2* are thus equivalent. Likewise, states *S1* and *S3* are equivalent for the same reason. Thus, we can redraw the FSM as in Figure 6.35(b). The FSMs in Figure 6.35(a) and (b) have exactly the same behavior—for any sequence of inputs, the two FSMs provide exactly the same sequence of outputs. If we encapsulate the FSM as a box as in Figure 6.35(c), the outside world cannot distinguish between the two FSMs based on the outputs.

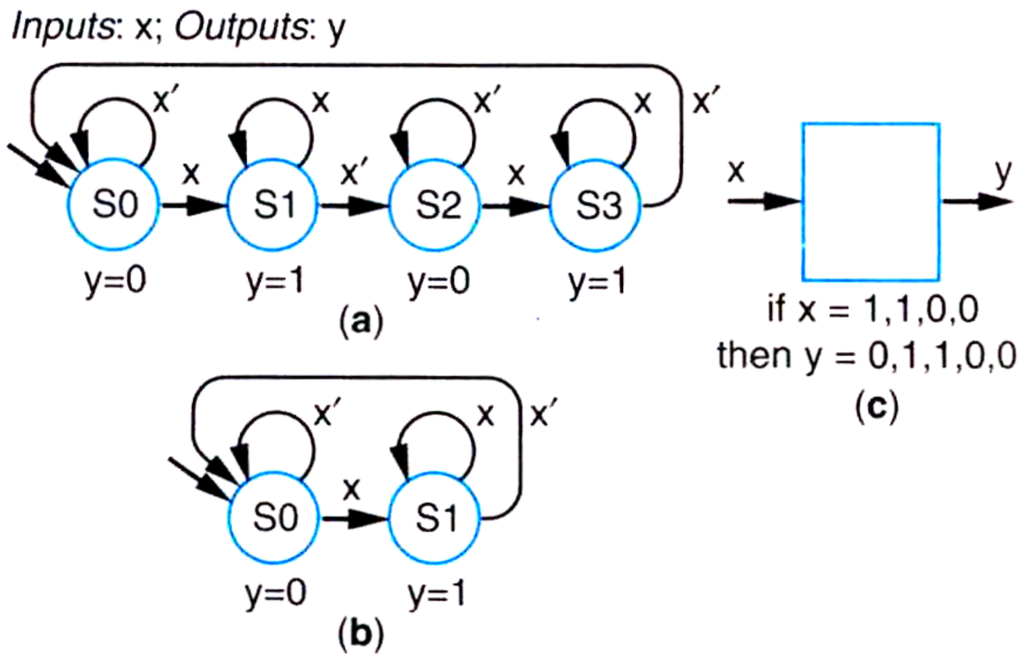


Figure 6.35 Eliminating redundant states: (a) original FSM, (b) equivalent FSM with fewer states, (c) the FSMs are indistinguishable from the outside, providing identical output behavior for any input sequence.

Two states are equivalent if:

- they assign the same values to outputs, AND
- for all possible sequences of inputs, the FSM outputs will be the same starting from either state.

For large FSMs, visual inspection cannot guarantee that we've removed all redundant states—a more systematic approach is needed, which we now introduce.

Implication Tables

Intuitively, we know that two states cannot be equivalent if they produce different outputs for the same sequence of inputs. Consider the FSM in Figure 6.36, which is almost identical to the FSM in Figure 6.35 with a slight modification — in state *S2*, the FSM now outputs y=1 instead of y=0. States *S0* and *S2* therefore clearly are not equivalent, because they have different output values. States *S1* and *S3* produce the same output, but when we transition from either state to the corresponding next state, the output differs. For example, if the FSM starts in state *S1* and x becomes 0, the next state (*S2*) outputs y=1, but if

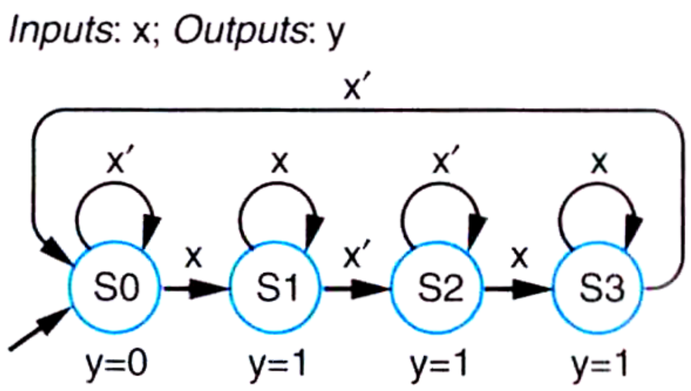


Figure 6.36 A variant of the FSM in Figure 6.35—states *S0* and *S2* cannot be equivalent because they output different values, and states *S1* and *S3* can't be equivalent because they have nonequivalent next states for the same input values.

the FSM had started in *S3*, the next state (*S0*) would output y=0. Thus, *S1* and *S3* cannot be equivalent, because the same input sequence results in a different output sequence.

If two states' outputs are not equivalent, the two states clearly are not equivalent. Furthermore, if two states' *next states* are not equivalent for a given input value, then the two states are also not equivalent. Using these concepts of nonequivalent states, Table 6.2 describes an algorithm for reducing an FSM's number of states.

TABLE 6.2 Algorithm for state reduction.

	Step	Description
1	*Mark state pairs having different outputs as nonequivalent*	States having different outputs obviously cannot be equivalent.
2	*For each unmarked state pair, write the next state pairs for the same input values*	
3	*For each unmarked state pair, mark state pairs having nonequivalent next-state pairs as nonequivalent. Repeat this step until no change occurs, or until all states are marked.*	States with nonequivalent next states for the same input values can't be equivalent. Each time through this step is called a ***pass***.
4	*Merge remaining state pairs*	Remaining state pairs must be equivalent.

When comparing all possible pairs of states by hand, using a graphical table ensures that we don't miss any pairs. Consider the FSM of Figure 6.35(a). The FSM has 4 states, therefore there are $4^2 = 16$ possible state pairs. Figure 6.37(a) shows those possible pairs graphically in a table, with the states listed along the row and column headings. Each cell corresponds to a state pair. We can simplify the table size by removing redundant cells (e.g., row *S0*, column *S1* is the same as row *S1*, column *S0*) and removing meaningless cells along the diagonal of the table (state *S0* is obviously equivalent to state *S0*). The reduced table is shown in Figure 6.37(b).

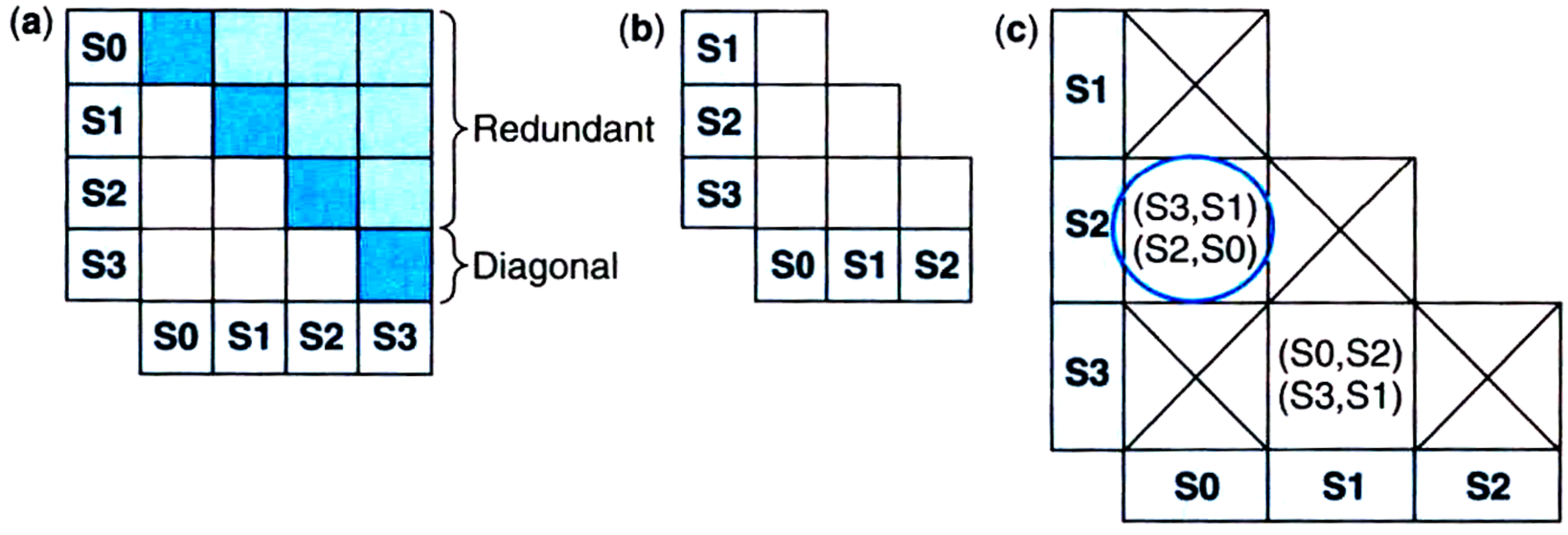


Figure 6.37 Table of state pairs: (a) original table comparing all pairs, (b) simpler table comparing only unique and relevant pairs, (c) after initial filling in with FSM state information.

Figure 6.37(c) steps through the state reduction algorithm of Table 6.2 for the FSM of Figure 6.35(a).

Step 1 involves looking at every table cell and marking that cell with a large "X" if the states for that cell have different outputs. We refer to such cells as being *marked*. The first state pair (*S1*,*S0*) is not equivalent because *S0* outputs y = 0, while *S1* outputs y = 1. We then look at state pair (*S2*,*S0*), (*S2*,*S1*), and so on, and finally (*S3*,*S2*), marking state pairs having different outputs, resulting in the Xs shown in Figure 6.37(c).

Step 2 involves writing the next state pairs for each remaining unmarked cell. There are two unmarked cells:

- (*S2*,*S0*) (circled in Figure 6.37(c)): When x=1, state *S2*'s next state is *S3*, while state *S0*'s next state is *S1* (we see this by looking at the FSM in Figure 6.35(a)). Thus, we write "(*S3*,*S1*)" in that cell (the order doesn't matter), meaning that for states *S2* and *S0* to be equivalent, *S3* and *S1* must be equivalent. We then consider the case when input x=0, in which case the next states are *S2* and *S0*, so we write "(*S2*,*S0*)" in that cell also.
- (*S3*,*S1*): When x=0, the next states are *S0* and *S2*, so we write (*S0*,*S2*) in the cell. For x=1, we write (*S3*,*S1*) in the cell.

Step 3 involves marking as nonequivalent any unmarked cells whose next state pairs are already marked as nonequivalent. Looking at cell (*S2*,*S0*), the next state pair (*S3*,*S1*) is not marked, nor is next state pair (*S2*,*S0*) (which happens to be the current cell), so we can't mark this cell. Likewise, for cell (*S3*,*S1*), the next state pair (*S0*,*S2*) is not marked, nor is the next state pair (*S3*,*S1*), so we can't mark this cell.

Because we made a pass through step 3 without any changes, we don't repeat step 3, and instead move on to step 4.

Step 4 involves declaring the unmarked state pairs as equivalent, so *S2* and *S0* are equivalent, and *S3* and *S1* are equivalent. To finalize step 4 of the algorithm, we combine the equivalent states in the FSM. After combining states *S2* and *S0*, and combining states *S3* and *S1*, we obtain the FSM in Figure 6.35(b).

The method we have just employed is known as the ***implication table*** method for state reduction.

Naturally, not every FSM can have its number of states reduced. For example, let's use the implication table method on the FSM in Figure 6.36. With 4 states, the FSM's implication table will be the same size as the previous example, as shown in Figure 6.38(a). **Step 1** marks state pairs with different outputs, shown in Figure 6.38(a). **Step 2** lists, for each unmarked cell, the next state pairs for identical input values, as also shown in Figure 6.38(a).

In **step 3's first pass**, we first examine the cell for state pair (*S2*, *S1*). Naturally, the next state pair (*S2*, *S2*) is equivalent. The next state pair (*S3*, *S1*) is unmarked, so we cannot mark (*S2*, *S1*). We then examine the cell for state pair (*S3*,*S1*), and find that the next state pair (*S0*,*S2*) has its cell marked. This tells us that *S3* and *S1* cannot be equivalent (because they could transition to nonequivalent states for the same input values), so we mark the cell for *(S3,S1)*. Similarly, we mark (*S3*,*S2*) since its first next state pair, (*S0*,*S2*), has its cell marked. Completing step 3's first pass results in the table of Figure 6.38(b).

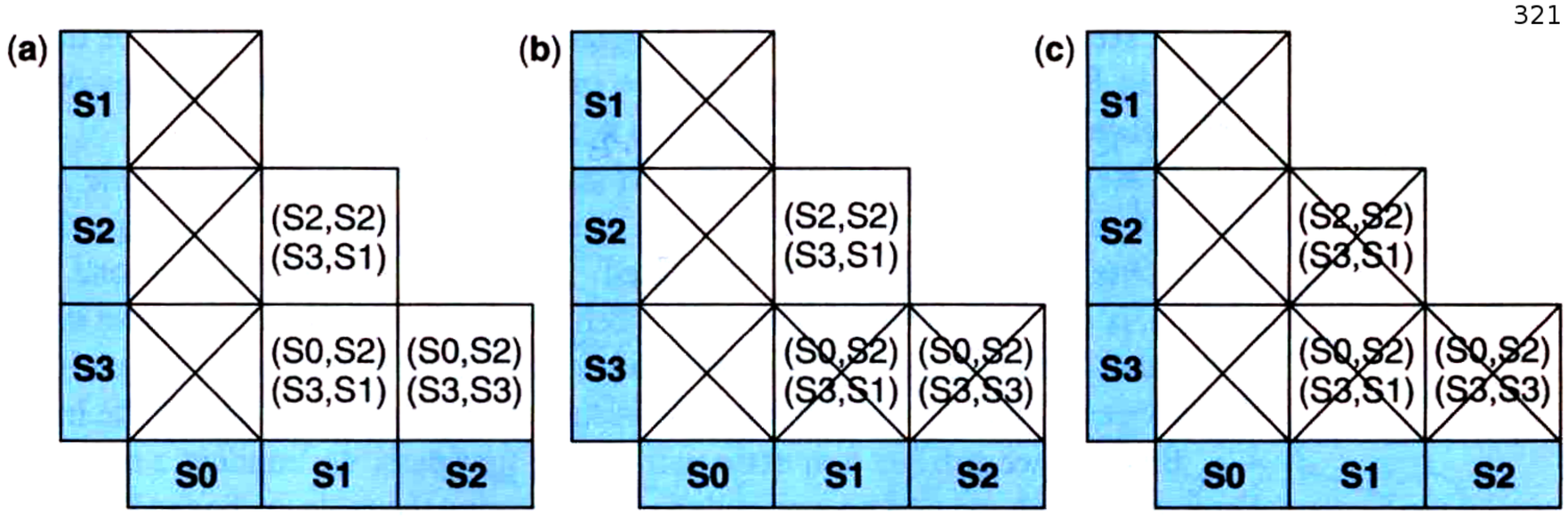


Figure 6.38 Implication table for FSM in Figure 6.36: (a) table after initial setup and steps 1 and 2, (b) after step 3's first pass through the table, (c) after step 3's second and final pass through the table.

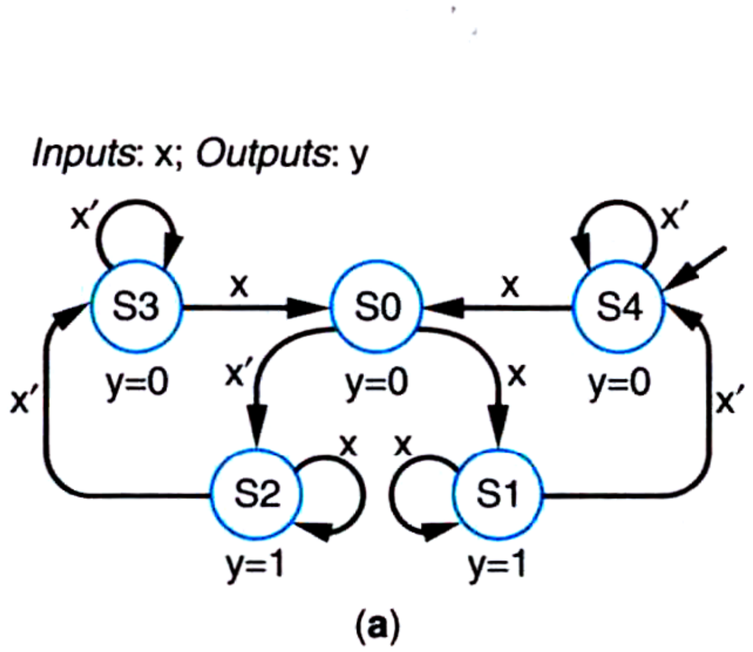
Because the table changed during the first pass (we marked two state pairs), we must make a **second pass**, because changes in the table may affect state pairs that we already looked at and left unmarked. In the second pass, we again look at state pair (*S2*,*S1*). Naturally, the next state pair (*S2*,*S2*) is equivalent. The next state pair (*S3*,*S1*), however, is now marked, and therefore we mark (*S2*,*S1*).

With all pairs in the table marked, as seen in Figure 6.38(c), we can conclude that no states in the FSM are equivalent, and thus we leave the FSM unchanged.

We now provide another example of state reduction.

EXAMPLE 6.13 Minimizing states in an FSM using an implication table

Consider the FSM in Figure 6.39(a). Unlike previous examples, this FSM has 5 states, resulting in more possible state pairs than in previous examples. The first task in minimizing the FSM's states is to construct an implication table so we can compare every state with each other as a state pair.



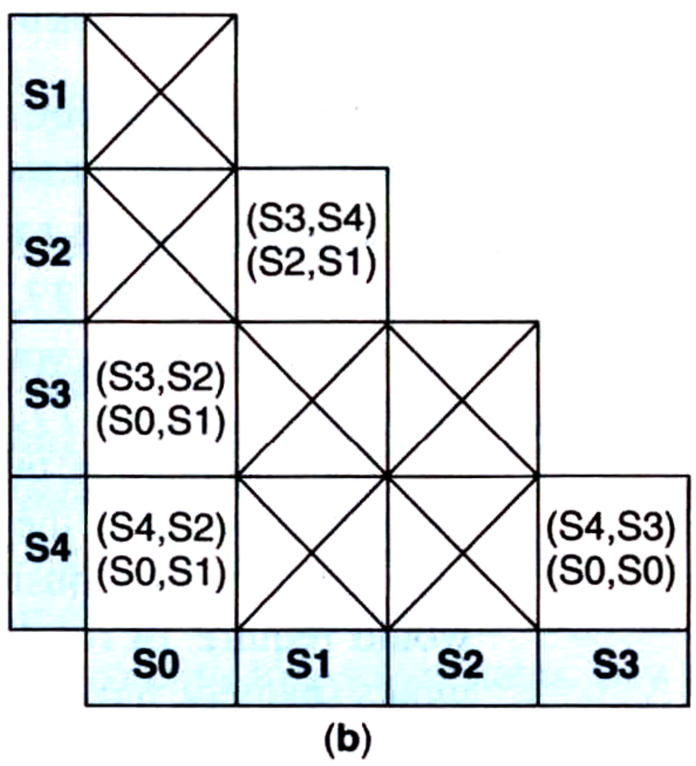


Figure 6.39 An FSM needing state reduction: (a) original FSM, (b) implication table after steps 1 and 2.

In **step 1** of our state reduction algorithm, we mark with an X state pairs that we can easily tell are not equivalent because their outputs differ, as shown in Figure 6.39(b).

In **step 2**, we write in all the next state pairs for unmarked cells of the implication table, as shown in Figure 6.39(b). Since there are only two possible combinations of inputs (either `x=0` or `x=1`), each unmarked cell will have two next state pairs.

In **step 3's first pass**, we mark each state pair if one of their next state pairs is marked. During our first pass through the table, we will examine four state pairs. Starting with (*S2,S1*), we see that both of its next state pairs are unmarked. Looking at (*S3,S0*), we see one of its next state pairs, (*S3,S2*), is marked, so we mark (*S3,S0*)'s cell. We also mark (*S4,S0*) because its next state pair (*S4,S2*) is marked. We leave (*S4,S3*) unmarked as both of its next state pairs are unmarked, thus completing the first pass. Figure 6.40(a) reflects the results of our first pass through the implication table.

Because we marked new state pairs in the first pass, we conduct a **second pass** through step 3. During that pass, we find no new cells to mark, leaving the table unchanged. We thus move on to step 4.

In **step 4**, we declare the unmarked state pair (*S2,S1*) as equivalent, and the unmarked state pair (*S4,S3*) as equivalent. We combine states *S2* and *S1*, and we combine states *S4* and *S3*, resulting in the new FSM shown in Figure 6.40(b). Note that the two transitions with conditions `x'` and `x` from *S0* could be replaced by one transition with no conditions.

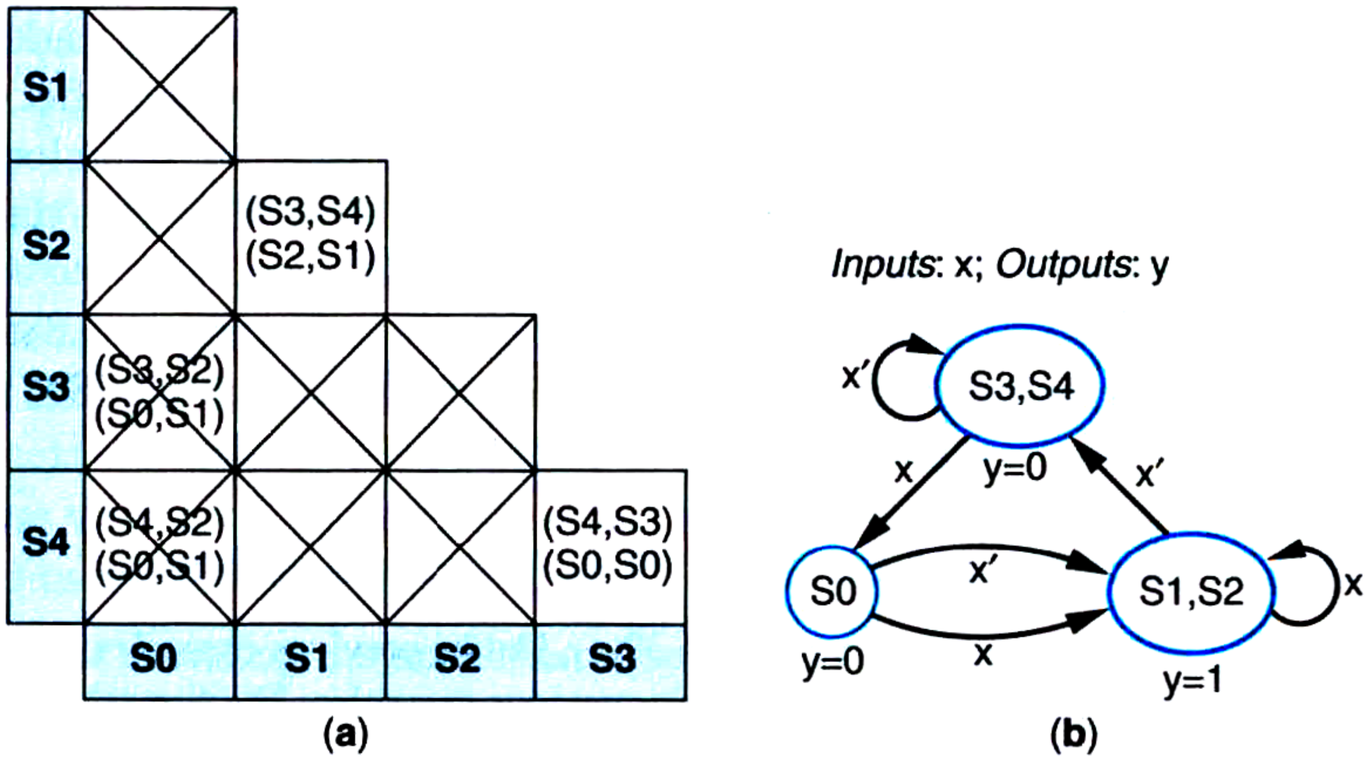


Figure 6.40 Implication table and minimized FSM: (a) implication table after first pass, (b) minimized state machine with states *S1* and *S2* combined, and *S3* and *S4* combined.

In this example, by reducing the number of states from 5 down to 3, we have reduced the minimum state register size from 3 bits down to 2 bits, perhaps reducing circuit size.

Sometimes equivalent states may overlap. For example, assume that for some FSM with states {*T0*, *T1*, *T2*, *T3*, *T4*}, you find that state pairs (*T0,T1*), (*T1,T2*) and (*T2,T0*) are equivalent. How do you deal with the overlapping equivalencies? The answer is simple: the three states, *T0*, *T1*, and *T2* can be combined into a single state.

The implication table method is suitable for hand-optimizing small FSMs such as those introduced in the previous examples, but can quickly become unwieldy for FSMs with more states. Consider the 15-state FSM in Figure 6.41. Its reduced implication table would require 14 rows and 14 columns, and 105 state pairs. With two combinations of inputs (namely, `a=0` or `a=1`), each state pair would have two next state pairs, and, in the worst case, we would need to check 105*2=210 next state pairs during our first pass alone. What if the same FSM had four inputs (say, `a`, `b`, `c`, and `d`) instead of one? With four inputs, there would be 4^2=16 combinations of inputs (i.e. `a'b'c'd'`, `a'b'c'd`, `a'b'cd'`, ..., `abcd`) and up to 16 next state pairs in each cell in the implication table. If instead the FSM had, say, 100 states (a reasonable number), the implication table would have on the order of 100*100 = 10,000 state pairs.

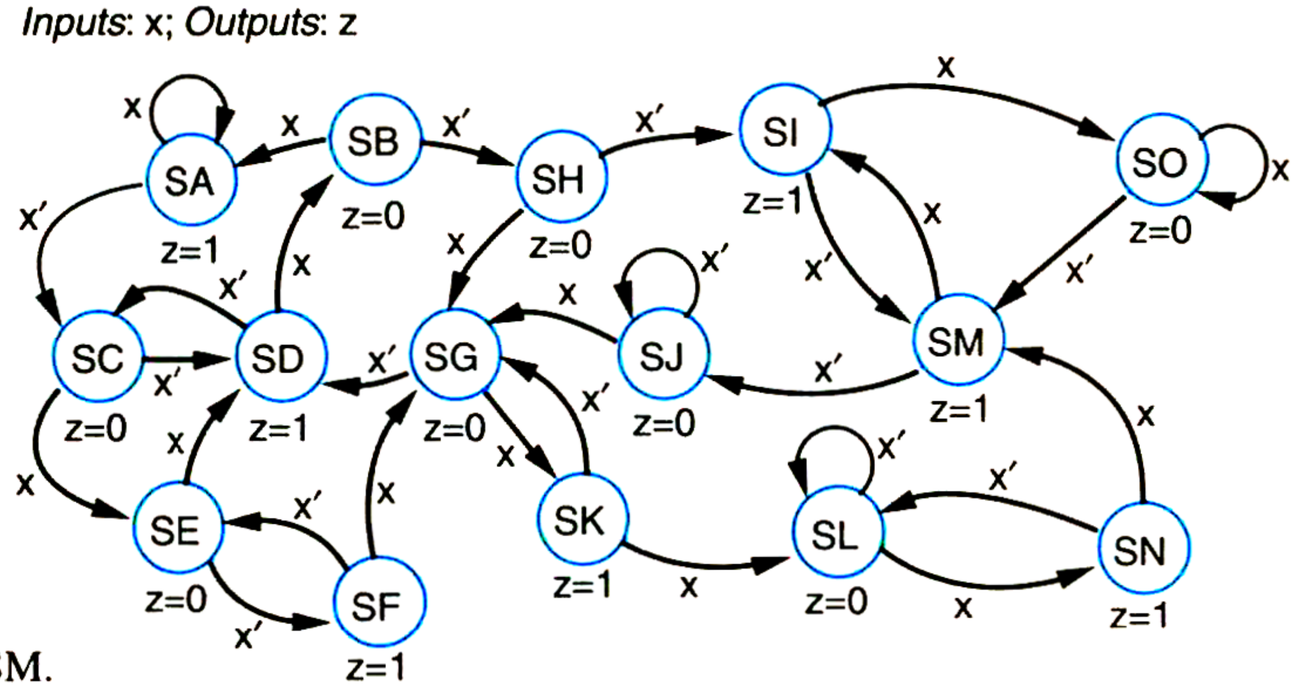


Figure 6.41 A 15-state FSM.

State reduction is therefore typically performed using automated tools. For smaller FSMs, the tools may implement the implication table method. For larger FSMs, the tools may need to resort to heuristics to avoid inordinately large table sizes or numbers of next state pairs.

Even when we reduce the number of states, we are not guaranteed that such state reduction actually reduces the size of the resulting logic. One reason is because reducing the states might not reduce the number of required state register bits—reducing the states from 15 down to 12 does not reduce the minimum state register size, which is four in either case. Another reason is because, even if the state reduction reduces the state register size, the combinational logic size could possibly *increase* with a smaller state register, due to the logic having to decode the state bits. Thus, automated state reduction tools may need to actually implement the combinational logic before and after state reduction, to determine if state reduction ultimately yields improvements for a particular FSM.

State Encoding

State encoding is the task of assigning a unique bit representation for each state in an FSM. Some state encodings may optimize the resulting controller circuit by reducing circuit size, or may trade off size and performance in the circuit. We now discuss several methods for state encoding.

Alternative Minimum-Bitwidth Binary Encodings

Previously, we assigned a unique binary encoding to each state in an FSM using the fewest number of bits possible, representing a ***minimum-bitwidth binary encoding***. If there were four states, we used two bits. If there were five, six, seven, or eight states, we used three bits. The encoding represented the state in the controller's state register. There are many ways to map minimum-bitwidth binary encodings to a set of states. Say we are given four states, *A*, *B*, *C*, and *D*. One encoding is *A*:00, *B*:01, *C*:10, *D*:11. Another encoding is *A*:01, *B*:10, *C*:11, *D*:00. In fact, there are 4*3*2*1 = 4! = 24 possible encodings into two bits (4 encoding choices for the first state, 3 for the next state, 2 for the next, and 1 for the last state). For eight states, there are 8!, or over 40,000, possible encodings into three bits. For *N* states, there are *N!* (*N* factorial) possible encodings—a huge number for any *N* greater than 10 or so. One encoding may result in less