Tutorial 5

Sequential Circuits - Finite State Machines

THINGS TO LOOK FOR...

- The extension of sequential systems to include input and output capability.
- Guidelines for the design of finite state machines.
- The Mealy and Moore machines and their differences.
- Assigning values to state variables – several approaches.
- Developing the defining equations for a finite state machine.
- Modeling FSMs using RTL and structural models.
- Optimizing finite state machines using row reduction and implication charts.

5.0 INTRODUCTION

In this tutorial, we assume a background in basic memory devices and how such devices can be utilized in the design and implementation of elementary sequential circuits such as counters, dividers, and timers. We now build on that background as we examine the development of general purpose sequential machines. Such machines find application in controlling and coordinating the activity of most modern microprocessors, microcontrollers, and microcomputers as well as more complex digital systems. We will also develop both RTL and structural level models in Verilog for the FSM.

Design is one step in the process of developing embedded applications. Equally important as an integral part of that process is the optimization of the system to increase reliability, and to reduce power, cost, and weight. In this tutorial we will introduce and examine several different approaches for reducing the complexity of sequential machines.

5.1 Finite State Machines

Normally, the input and output capabilities of counting, timing, and dividing sequential machines are limited to a few control lines, an output frequency, or a signaling event. To be able to perform the complex computation and control necessary in modern digital systems, we must accept a greater variety of input signals and develop a richer set of output capability.
In our studies, we will use the model for the finite state machine given by the 5-tuple,

\[ M = (I, O, S, \lambda, \delta) \]

- \( I \) - Finite nonempty set or vector of inputs
- \( O \) - Finite nonempty set or vector of outputs
- \( S \) - Finite nonempty set or vector of states
- \( \delta \) - Mapping \( I \times S \rightarrow S \)
- \( \lambda_1 \) - Mapping \( I \times S \rightarrow O \) - Mealy Machine
- \( \lambda_2 \) - Mapping \( S \rightarrow O \) - Moore Machine

Let’s now take a detailed look at how we can begin to apply such a model to the design of more sophisticated and capable finite state machines.

5.2 The Design of a Serial Pattern Detector

We’ll begin with a simple pattern or sequence detector. One finds possible uses for such a system in telecommunications systems or in other applications where it may be necessary to synchronize or lock onto a data stream containing a preamble or initial synchronizing pattern.

For this design, we assume that data will enter the system, in serial, one bit at a time and that the specifications require detecting the preamble pattern 1010 in the data. If the pattern is recognized, the system is to issue an output signal, \textit{found}. The detector’s output signal will be a function of both the input and the current state. Thus, our design will implement a Mealy machine.

5.2.1 The High Level Design

We begin the design with a high level outside view of the system. The details of the larger system environment, which may be a complex telecommunications system, are not important; the public interface that the pattern detector presents to that environment is. The enclosing system provides a serial stream of data and expects an annunciating event if a specified pattern is detected. The system requirements are given as,

\textbf{Inputs}

One input – A serial stream of data. The data is expressed in big endian format with the least significant bit received first.

\textbf{Outputs}

One output – A logical 1 that will annunciate the presence of each occurrence of the sequence 1010 in the data stream.

\textbf{Algorithm}

The pattern detection is to use a sliding window algorithm.

As a first step, we capture the requirements from the outside point of view in a high level block diagram as shown in Figure 5.0. In this case, the diagram is rather elementary.
We can execute a second level of decomposition or refinement, as we see in Figure 5.1, to distinguish the pattern recognition portion of the system and the output section. There is little need to decompose further.

5.2.2 The State Diagram

To formally capture the behavior of the system, we begin with a state diagram. The objective is to be able to express the behavior of the system in time. We must identify each legal state of the system and each of the transitions amongst those states.

States

Each state in the machine corresponds to a vertex, or node, in the state diagram. If the system has ten states, the state diagram will have 10 nodes.

Transitions

For each state or vertex in the state diagram, there will be $2^p$ directed arcs, one for each combination of inputs. The arcs identify the state transitions caused by the input variables. These arcs express the mapping $I \times S \rightarrow S$ we saw in the earlier finite state machine model.

If the system has three inputs, each state will have $2^3$ arcs leaving; one arc for each of the eight possible input combinations. It’s clear that such a combinational explosion is one of the limitations of the basic finite state machine.

In the state diagram for either machine, we label each transition arc as shown.

The input portion of the label identifies the causative vector of input values associated with the transition. For the Mealy machine, the label also identifies the resulting vector of output values. For the Moore machine, the output information is written inside the node, usually in square brackets.

Behavior

The state diagram completely describes the system. It shows the succession of states through which the sequential machine passes and the corresponding output sequence that it produces. We distinguish two of those states,

Initial State

State of the machine prior to the application of any input sequence. The initial state is normally the power ON state of the system; it is also called the idle state or the quiescent state.
**Final State**

State of the machine after the application of the last input sequence.

**Prelude to the Design**

Before starting the design, the specification must be clarified. For the following sample sequence …10101010….there are two possible interpretations

1. The data stream can be examined in four bit groups.
   - If the pattern is detected, the detector begins a new search starting with the next group of four bits.
   - For the above sequence, the pattern will be detected twice,
     
     ....1010 1010 ....
     ....1010 1010 ....
   
2. Part of the pattern can be reused.
   - For the above data stream, the pattern will be detected three times,
     
     ....1010 1010 ....
     ....101010 10....
     ....1010 1010 ....

Such an interpretation is called a *sliding window*. For this design, as required by the specification, the system will implement the second interpretation.

**Beginning the Design**

The system will start in the *idle* state. Most of designs use an initial or idle state. This is also the state that the system enters following a power ON or other form of reset condition. In this case, we will label that state as *state A*.

Since there is only a single input, there can only be two possible transitions from each state node. In the idle state, the system is at time t₀; the input signal can be either a 0 or a 1.

If the input bit is a logical 0, the state diagram will reflect that fact with a transition to a new state, *state B*. *State B* captures several pieces of information / history about the system. By virtue of being in a new state, B, we know,

1. The pattern detection process has started.
2. Based upon the specification and from the meaning of *state B*, it is known that 1 bit of the pattern being searched for is correct – that bit is a logical 0.
3. The pattern has not yet been found; thus, the output must be false.

If the input data bit is a logical 1

1. The state machine remains in *state A*. Again, based upon the specification and from the meaning of *state A*, it is known that no bits in the pattern have been correctly recognized.
2. The pattern has not yet been found; thus, the output must be false.

These two possibilities cover all input combinations for *state A* as shown in Figure 5.2. The state diagram becomes,
Observe that each transition is labeled showing causative input vector and resulting output vector.

In state B, the system is at relative time $t_1$. Again the input can be either a 0 or a 1.

If the input data bit is a 0
1. The system remains in state B. The implications of state B still hold, there has only been one correct bit received.
2. The system remains at relative time $t_1$.
3. The output of the system is false.

If the input data bit is a 1, the system enters a new state, state C. By virtue of being in state C, it is known,
1. That 2 bits of the pattern are correct – that subsequence is '0 1'.
2. The pattern has not been found yet; thus, the output must be false.

The state diagram is now show in Figure 5.3.

In state C, the system is now at relative time $t_2$. Once again the input can be either a 0 or a 1,

If the input data bit is a 0, the system enters a new state, state D. By virtue of being in state D, it is known,
1. That 3 bits of the pattern being searched for have been recognized – that subsequence is '0 1 0'.
2. The pattern has not been found yet; the system is now looking for the fourth data bit, thus, the output must be false.

If the input data bit is a 1
1. The detection process has failed and must start over. The system must return to state A.
2. The system is now at relative time $t_0$.
3. The output of the system is false.

The state diagram in Figure 5.4 reflects the state of the system after the third bit is entered.
In state D, the system is at relative time t₃. Once again the input can be either a 0 or a 1.

If the input data bit is a 0, the system must return to state B. By virtue of being in state B, we know,
1. The implications of state B still hold. The two successive logical 0 bits mean that there has only been one correct bit received the last logical 0 bit.
2. The system remains at relative time t₁.
3. The output of the system is false.

If the input data bit is a 1, the system returns to state C and the output is a logical 1
1. It is known that 4 bits of the pattern being searched for are correct – that subsequence is '0 1 0 1'.
2. It is also known that the last two bits of the pattern just received correctly match the initial two bits of the second potential reception of the pattern. That information is implicit in state C.
3. The system is now at relative time t₂.
4. The output of the system is true.

The final state diagram is now presented in Figure 5.5,

![State Diagram](image)

Figure 5.5
Pattern Detector
State Diagram After Four Bits Entered

The master (or Power ON) reset which, on application, forces the system into the initial state is included; the complete behavior of the system has now been captured. Such a diagram can be used for further discussion and analysis. The state table is a better vehicle to begin the next stage of design

### 5.2.3 The State Table

The state table contains the same information as state diagram simply rewritten in tabular form. The state tables we’ve written in another tutorial and in the text have had two major subdivisions: one column showing the state of the system at time tₙ, the present state of the system, and a second column showing the state of the system at time tₙ₊₁, the next state of the system.

The next state subdivision is now refined to incorporate the mapping \( I \times S \rightarrow S \), that is, to reflect the behavior of the system for each possible combination of system inputs. To this end, under the next state heading, the state table will contain \( p \) columns; one for each of the \( p \) combinations of the input symbols in set \( I \).

The state table will contain \( n \) rows, one for each possible state in the set \( S \) (in the system). The cross product of the columns and rows will give all possible combinations of
inputs and states. To implement the state machine will require \( k \) memory elements; \( k \) is smallest integer value such that \( k \geq \log_2 p \).

For the current design, the next state category will have two columns: one for the input data taking a value of logical 0 and one for the input data taking a value of logical 1. The state machine will have 4 rows, one for each state. Taking the information directly from the state diagram in Figure 5.5 gives the state table in Figure 5.6.

<table>
<thead>
<tr>
<th>Present State ( t = t_n )</th>
<th>Next State ( t = t_{n+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 0 )</td>
<td>( x = 1 )</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 5.6
Pattern Detector State Table

5.2.4 The Output Table

The next step in the process is to specify the outputs from the system; these appear in an output table. Once again this information can be extracted directly from the state diagram. Often the state and output tables are merged into single table.

The output table for a Mealy machine also has \( p \) columns, one for each combination of input symbols in set \( I \) and \( n \) rows, one or each state in the set \( S \). From the earlier model, the output table gives the output matrix,

\[
\begin{align*}
I \times S & \rightarrow O \quad \text{Mealy Machine} \\
S & \rightarrow O \quad \text{Moore Machine}
\end{align*}
\]

Each combination of the elements of the input vector with the present state specifies the output of system. If the system is to implement a Moore machine, the output entries are independent of the input. Once again reading directly from the state diagram, the output table for the current design is given in Figure 5.7. This design implements a Mealy machine.

<table>
<thead>
<tr>
<th>Present State ( t = t_n )</th>
<th>Next State ( t = t_{n+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 0 )</td>
<td>( x = 1 )</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.7
Pattern Detector Output Table
5.2.5 The State Assignment

In the earlier finite state machines that we have designed in the text, the values for the state variables in each state were implicit in the specified counting or dividing sequence. Such is no longer the case. As designers, it’s now our responsibility to choose the system’s state variables and assign appropriate combinations to each state in such a manner that we can uniquely identify each state in the system. Selecting a state variable assignment is an important step in the design of any finite state machine. There are a wide variety of techniques, with varying degrees of complexity, that are available for making such assignments. Let’s look at several.

Binary Assignment

The easiest method is to simply use a binary sequence. The initial state gets the pattern of binary 0; each subsequent state gets the next binary number. The approach often works without problems. The one major limitation is building output signals as collections of combinational expressions; that is, the outputs of the combinational logic circuits are subject to race conditions and hazards. With such an approach, one always runs the risk of having decoding spikes on systems outputs.

One Hot Code

The one hot code makes a trade-off between increased circuit complexity and simplicity in generating output signals. With the exception of the initial state, the one hot code utilizes state assignments with only a single one bit in each. For example, a four state machine might utilize an assignment such as that given in Figure 5.8.

Certainly the four states could have been uniquely identified using only two bits. The three bit pattern, however, eliminates the need to decode any state or input combinations to produce an output signal thereby also clearly eliminating any logic hazards. When bit C is a 1, it is known that the machine must be in state 1 and vice versa.

Gray Assignment

The purpose of a Gray code assignment is also to address the problem of race conditions producing hazards when decoding output signals. With a Gray code assignment, the goal is to ensure that we have only a single variable change between states.

The first step in the approach is to develop a table listing all states. For each state, identify the previous and the next states. Such states are then assigned state variable patterns such that, in the ideal case, previous to present and present to next transitions involve only a single variable change. For example, the sequence

‘1 0 0’ → ‘1 0 1’ → ‘1 1 1’

illustrates such a transition. By using such an assignment, one can ensure that the transitions between the three states follow a Gray code sequence and that if any of the states are decoded as an output signal, there will not be any hazards.

To formulate a Gray code assignment for the current design, we perform the following sequence of steps,
1. Identify the adjacent state transitions.
   These are identified in Table 5.0,

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A,D</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B,D</td>
<td>C</td>
<td>A,D</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>B,C</td>
</tr>
</tbody>
</table>

2. Assign states such that there is a single variable change from present to next and previous to present. A Karnaugh map is a rather useful graphical aid.

   Satisfying the constraint for all states may not always be possible. In such cases, state sequences that contribute to combinational relations of output signals should be given preference.

   For the system under design, there are 4 states; therefore the assignment will require 2 state variables.

   A common heuristic suggests assigning the all 0’s pattern to the initial state. The motivation for doing so is that at Power ON, normally all storage elements (exclusive of RAM) are reset to ‘0’. Once the initial state is specified, the rest of the states are assigned so as to meet the state adjacency requirements identified earlier as we see in the adjacent Karnaugh map. From the map in Figure 5.9, one such assignment is now given as that in Figure 5.10.

   5.2.6 The Transition Table

   Once the state assignment is determined, the next step is to use that assignment to construct a transition table. The transition table combines the state table with the state assignment by substituting the state variable combination for each state back into the state table.

   For the current design the transition table is given in Figure 5.11.
5.2.7 The Input Equations

Using information from the transition table, the K Maps given in Figure 5.12 are now written for the flip-flop input equations.

As discussed in the text, the $\alpha$ and $\beta$ notation reflects a 0 to 1 and a 1 to 0 transition of the state variable respectively.

If the system is to be implemented using JK flip-flops, the logic equations can be written directly from the maps as given in Figure 5.13.
5.2.8 The Output Equations

The output equations follow in a similar manner; once again, a K Map used. For the current design we have the K Map and output equation given in Figure 5.14.

\[
\begin{array}{c|cc|c}
  MN & 0 & 1 & x \\
  \hline
  00 & 0 & 0 & \text{found} = M \cdot \overline{N} \cdot X \\
  01 & 0 & 0 & \\
  11 & 0 & 0 & \text{found} \\
  10 & 0 & 1 & \\
\end{array}
\]

Figure 5.14
Pattern Detector Output Equation

The JK flip-flop is first implemented using a behavioral model in Figure 5.15.

```
module JKFF(q, qBar, J, K, clk, por);
  input J, K, clk, por;
  output q, qBar;
  parameter delay0 = 2; // delay reset to q
  parameter delay1 = 3; // delay clock to q
  parameter delay2 = 2; // delay for qBar with respect to q
  reg q;

  not #delay2 n1 (qBar, q);
  always@ (negedge por or posedge clk)
  begin
    if(por==0)
      #delay0 q = 0;
    else if ((J == 0) && (K==1))
      #delay1 q = 0;
    else if ((J == 1) & (K==0))
      #delay1 q = 1;
    else if ((J == 1) & (K==1))
      #delay1 q = qBar;
  end
endmodule
```

Figure 5.15
J K Flip Flop Behavioral Model
Next, the structural model for the design using JK flip-flops follows in Figure 5.16.

```verilog
module PatternRecog0(foundIt, qm, qn, dataIn, clk, por);
    // build the pattern recognizer
    // declare the inputs and outputs
    input       dataIn, clk, por;
    output      foundIt, qm, qn;

    // implement the state machine
    and   and0(jm, qn, dataIn);
    xnor  xnor0(km, qn, dataIn);
    JKFF  M(qm, qmBar, jm, km, clk, por);
    not   inv0(ndataIn, dataIn);
    or    or0(jn, qm, ndataIn);
    JKFF  N(qn, qnBar, jn, qm, clk, por);

    // output equation
    and   and1(foundIt, qm, qnBar, dataIn);
endmodule
```

The code module in Figure 5.17 provides an RTL-behavioral implementation of the pattern detector.

```verilog
module PatternRecog0(foundIt, state, dataIn, clk, por);
    // declare the inputs and outputs
    input       dataIn, clk, por;
    output      foundIt;
    output[1:0] state;
    reg         foundIt;
    reg[1:0]    state;

    // define parameter or name of each state
    parameter A = 2'b00;
    parameter B = 2'b01;
    parameter C = 2'b11;
    parameter D = 2'b10;

    // build the pattern recognizer
    always@ (negedge por or posedge clk)
    begin
        // reset the machine
        if(por==0)
            begin
                assign state = A;
                assign foundIt = 1'b0;
            end
        else
            // continued
    end
endmodule
```
// implement the state machine
begin
    case(state)
        // initial state 0 correct pattern xxxx
        A:
            begin
                if(dataIn == 1'b0)
                    assign state = B;
                else
                    assign state = A;
                    assign foundIt = 1'b0;
            end
        // one correct pattern xxx0
        B:
            begin
                if(dataIn == 1'b0)
                    assign state = B;
                else
                    assign state = C;
                    assign foundIt = 1'b0;
            end
        // two correct pattern xx10
        C:
            begin
                if(dataIn == 1'b0)
                    assign state = D;
                else
                    assign state = A;
                    assign foundIt = 1'b0;
            end
        // three correct pattern x010
        D:
            begin
                if(dataIn == 1'b1)
                    begin
                        // four correct pattern 1010
                        assign state = C;
                        assign foundIt = 1'b1;
                    end
                else
                    assign state = B;
            end
    endcase
end
endmodule

Figure 5.17 cont.
Pattern Detector Behavioral Model
5.3 Design Guidelines

A general set of guidelines for designing finite state machines is now presented in Figure 5.18.

1. From a word description of the problem, form a state diagram.
2. From the state diagram develop the state table.
3. Check for redundant states.
4. Select a state assignment.
5. Develop the transition and output tables.
6. Develop a Karnaugh map for each state variable.
7. Select a memory device and develop the input equations from the Karnaugh map.
8. Develop a Karnaugh map for each output variable.
9. Develop the output equations from the Karnaugh map.

**Figure 5.18**
Design Guidelines for Finite State Machine Design

---

**Example 5.0**

Let’s look at another pattern recognition problem. The specification requires the design of a system that accepts a serial bit stream on the input and searches for either of the patterns: ‘011’ or ‘110’ in the input data stream. If, following the reception of three input data bits, the first pattern ‘011’ is found, the output `found0` is asserted high and if the second pattern ‘110’ is found the output `found1` is asserted high. Both outputs are to be in the low state otherwise.

The outside view of the system as expressed in the high level block diagram is given in Figure 5.19. In the block diagram, all inputs and outputs are identified.

**Figure 5.19**
High Level Block Diagram
System Detecting Two Patterns

From the design specification, the desired behavior of the system is captured in a state diagram. This is an essential first step.
Step 1

Figure 5.20 gives the state diagram for the detection system.

![Pattern Detector State Diagram](image)

The two paths for which the input sequence will yield a match are highlighted. After three input bits have been received, the system returns to the initial state.

Step 2

The state table, combined with the output table, is written from the state diagram and presented in Figure 5.21.

![Pattern Detector State and Output Table](image)
Observe that the design is a Moore machine. The output signals for all of the states are only a function of the present state, not the input: $S \rightarrow O$.

**Step 3**

We are skipping Step 3 for the moment.

**Step 4**

Select a state assignment. For this example, there are 10 states which require four state variables. Such an assignment will also give seven don’t care combinations.

A Gray assignment will be used to specify the values of the state variables. We begin by identifying the previous, present, and next state relationships. These are tabulated in Table 5.1.

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>1,2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3,4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>5,6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7,8</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7,9</td>
</tr>
<tr>
<td>3,4,5</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Based upon the transitions identified, we draw the K Map for the state assignment in Figure 5.22. Observe that the assignment is not unique. Further, we cannot completely ensure that all transitions have only a single variable change.
Step 5

Specify the transition and output tables. We accomplish this in Figure 5.23.

<table>
<thead>
<tr>
<th>Present State $t = t_n$</th>
<th>Next State $t = t_{n+1}$</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ABC$</td>
<td>$ABCD$</td>
<td>$ABCD$</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>1 0 0 0</td>
<td>2 0 0 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>3 0 0 1</td>
<td>4 0 0 1</td>
</tr>
<tr>
<td>2 0 1 0</td>
<td>5 0 1 0</td>
<td>6 0 1 0</td>
</tr>
<tr>
<td>3 1 0 0</td>
<td>7 1 0 0</td>
<td>7 1 0 0</td>
</tr>
<tr>
<td>4 0 1 1</td>
<td>7 0 1 1</td>
<td>8 0 1 1</td>
</tr>
<tr>
<td>5 1 1 0</td>
<td>7 1 1 0</td>
<td>7 1 1 0</td>
</tr>
<tr>
<td>6 0 1 0</td>
<td>9 0 1 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>7 1 1 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>8 0 1 0</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>9 0 1 1</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
</tr>
</tbody>
</table>

Figure 5.23
Pattern Detector Transition and Output Table

Step 6

Develop Karnaugh maps for each state variable. The K Map for state variable A is given in Figure 5.24.

<table>
<thead>
<tr>
<th>$ABC$</th>
<th>$00$</th>
<th>$01$</th>
<th>$11$</th>
<th>$10$</th>
<th>$Dx$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\alpha$</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>0</td>
<td>$\alpha$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>$\times$</td>
<td>$\times$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>$\alpha$</td>
<td>0</td>
<td>$\alpha$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>$\beta$</td>
<td>$\beta$</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>$\times$</td>
<td>$\times$</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

A map

Figure 5.24
K Map for State Variable A

The equation for $D_A$ follows,

$$D_A = A \cdot \overline{D} + B \cdot \overline{D} \cdot \overline{X} + B \cdot D \cdot \overline{X} + B \cdot \overline{C} \cdot D + \overline{A} \cdot B \cdot \overline{C} \cdot D \cdot X$$
Let’s now look at a system with more than one input. The system is designed to permit access to a restricted area if the user properly enters the correct access code. The access code for the system is the sequence 3 2 2 1 to open. All four digits must be entered correctly, in order, to allow access.

The high level block diagram for the system is given in Figure 5.25.

From the specification, the state diagram is drawn in Figure 5.26,
From the state diagram, the state and output tables follow naturally in Figure 5.27.

<table>
<thead>
<tr>
<th>Present State $t = t_n$</th>
<th>Next State $t = t_{n+1}$</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_2$ $X_1$</td>
<td>00 01 11 10</td>
<td>00 01 11 10</td>
</tr>
<tr>
<td>0</td>
<td>0 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 2 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 3 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

Figure 5.27
Combination Lock State and Output Table

For this problem, we’ll use a basic Gray sequence for the state assignment. Such an assignment gives the transition table in Figure 5.28.

<table>
<thead>
<tr>
<th>Present State $t = t_n$</th>
<th>Next State $t = t_{n+1}$</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_2$ $X_1$</td>
<td>00 01 11 10</td>
<td>00 01 11 10</td>
</tr>
<tr>
<td>A B</td>
<td>A B A B A B</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0 0 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>2 1 1</td>
<td>0 1 1 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>3 1 0</td>
<td>0 1 1 0</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

Figure 5.28
Combination Lock Transition Table

The K Maps for the two state variables then follow in Figure 5.29,
From which the flip-flop equations are given as,

\[ D_A = B \cdot \overline{X_1} \cdot X_2 \]

\[ D_B = \overline{A} \cdot X_2 \cdot (B \oplus X_1) \]

The RTL implementation of the circuit is given in the code module in Figure 5.30,

```verilog
module ComboLock0(open, state, comboIn, clk, por);
// declare the inputs and outputs
input clk, por;
input[1:0] comboIn;
output open;
output[1:0] state;
reg open;
reg[1:0] state;

// define parameter or name of each state
parameter s0 = 2'b00;
parameter s1 = 2'b01;
parameter s2 = 2'b11;
parameter s3 = 2'b10;
parameter true = 1'b1;
parameter false = 1'b0;
parameter zero = 2'b00;
parameter one = 2'b01;
parameter two = 2'b10;
parameter three = 2'b11;

// build the combination lock
```

Figure 5.29b
Combination Lock State Variable K Maps

Figure 5.30
Combination Lock RTL Model
// build the combination lock
always@ (negedge por or posedge clk)
begin
// reset the lock
    if(por==0)
        begin
            assign state = s0;
            assign open = false;
        end
// implement the combination lock
else
    begin
        case(state)
            // initial state 0 correct combination xxxx
            s0:
                begin
                    assign open = false;
                    if(comboIn == three)
                        assign state = s1;
                    else
                        assign state = s0;
                end

            // one correct combination xxx3
            s1:
                begin
                    if(comboIn == two)
                        assign state = s2;
                    else
                        assign state = s0;
                        assign open = false;
                end

            // two correct combination xx23
            s2:
                begin
                    if(comboIn == two)
                        assign state = s3;
                    else
                        assign state = s0;
                        assign open = false;
                end
        endcase
    end
end

Figure 5.30 cont.
Combination Lock RTL Model
5.4 State Machine Reduction

So far in the designs that have been presented, not much attention has been paid to optimization. Certainly, one could argue that with VLSI, FPGAs, Systems on a Chip, or large memory devices etc., why waste time trying to reduce or simplify a design. First, not all designs have such luxuries. Many consumer products stress simplification; the major reason is to save power. The fewer transistors the design uses, the lower the power consumption. Second, even with VLSI and the companion implementation technologies, if a design can be simpler, it is then possible to incorporate more functionality or features in the same real estate. A simpler design is easier to manufacture; even for VLSI. A third motivation is reliability; less complexity means fewer things to fail. A fourth consideration is weight. Whether the design is targeted to be flown in an aircraft or in a satellite or towards a consumer product, additional weight is generally considered to be a cost rather than a benefit.

We introduce several different simplification/optimization approaches. We’ll illustrate how the techniques are applied manually; however, they can be easily written as software design tools. Our focus will be on reducing the complexity of finite state machines.

The basic goal when trying to simplify such circuits and systems is to identify and to combine states that have equivalent behavior. Since the number of memory devices is directly related to the number of states, reducing the number of states can help to reduce the number of flip-flops. It’s important to keep in mind as we execute the process that the objective is to simplify the design; we are not making trade-offs. When we make trade-
offs, we are potentially eliminating functionality. When we optimize, we are keeping the functionality and improving the design.

Equivalent States are defined as those states which, for all input combinations, have the same set of output values and transition to the same or equivalent states under the same input values. We’ll examine two alternative algorithms

- Row matching
- Implication charts

We’ll begin with row matching.

5.4.1 The Row Matching Algorithm

The general approach under the row matching algorithm is rather straightforward.

1. Start with the state transition table.
2. Identify states with same output behavior.
3. If such states transition to the same next state, they are equivalent.
4. Combine into a single new renamed state.
5. Repeat until no new states are combined.

To illustrate the technique, we will use an application often found in telecommunications systems for error management, the generation of a parity check bit. The scheme is rather straightforward. A single logical 0 or logical 1 bit is appended to each data word or block. If odd parity is being used, the total number of one bits (including the parity bit) must be odd. If the data word has odd number of one bits—a logical 0 is added; if the data word has an even number of one bits—a logical 1 is added. If even parity is being used, the total number of one bits (including the parity bit) must even. If the data word has odd number of one bits—a logical 1 is added; if the data word has an even number of one bits—a logical 0 is added.

Table 5.2 illustrates the odd and even parity bits for a three bit data word.

<table>
<thead>
<tr>
<th>Original Data</th>
<th>Odd</th>
<th>Even</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Parity</td>
<td>Data Parity</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 0 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0 1</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1 0</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

We’ll implement the design of a system that accepts a serial data stream as input. On receipt, each incoming data bit is read from the stream and transferred to the output stream. After three data bits have been read and transferred, an even parity bit over the
three bits will be sent to the output and the cycle will repeat for the remainder of the message. Typical incoming and outgoing message streams (with even parity bits inserted) will appear as in Figure 5.31.

![Incoming Message Stream](image1)

![Outgoing Message Stream](image2)

**Figure 5.31**
Typical Data Stream With Even Parity Bits Inserted

With any design, it’s important to understand the requirements, scope, and complexity of the design before starting. Understanding the scope enables one to rapidly assess the solution at the end of the process and to identify potential problems along the path if the design appears to be becoming too complex.

For this system, there are two basic tasks:
1. Counting to three to know when to insert the parity bit and
2. Keeping track of whether an odd or even number of bits have come in.

Counting to three requires two flip-flops and keeping track of incoming bits requires one. Thus, the complexity of the design should be no worse than three flip-flops. The full state diagram for such a system is given in Figure 5.32. With fifteen states and therefore four flip-flops, the design appears to exceed the initial complexity estimate.

![Full State Diagram](image3)

**Figure 5.32**
Full State Diagram for an Even Parity Generator

From the state diagram, our first cut at the state table is given in Figure 5.33. In the state table, the rows are segregated into transitions that occur at \( t_0, t_1, t_2, \) and \( t_3 \). We now apply Steps 2..4 of the algorithm. We begin at time \( t_3 \) and work backwards. States that
occur at different times cannot combined otherwise the inherent history associated with each state in the machine will be lost. This means that states at times $t_2$, and $t_3$, cannot be combined, for example.

For the parity generation system, each of the states in the set \{7, 10, 12, and 13\} and each of the states in the set \{8, 9, 11, and 14\} have the same next state and same output. The states within each set can be combined into a single state without loosing any information.

All references to any states in the first set are replaced by $7a$ and all references to any states in the second set replaced by $8a$.

<table>
<thead>
<tr>
<th>Present State $t = t_n$</th>
<th>Next State $t = t_{n+1}$</th>
<th>Output Parity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x = 0$</td>
<td>$x = 1$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>$t_0$</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>$t_1$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>$t_2$</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>$t_3$</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.33
Full State Diagram for an Even Parity Generator
The state table now reduces to that given in Figure 5.34.

![Figure 5.34](image)

Partially Reduced State Diagram for an Even Parity Generator

The process is repeated for states at time $t_2$. In this case, states in the set $\{3, 6\}$ and in the set $\{4, 5\}$ have the same next states and outputs. Thus, they can also be combined.

The new state table is now given in Figure 5.35.

![Figure 5.35](image)

Partially Reduced State Diagram for an Even Parity Generator

The process is repeated once again at time $t_1$. Nothing can be combined here and thus the process terminates.
The final state table was given in Figure 5.35. The parity system has been reduced from original fifteen states to seven states and the number of flip flops has been reduced from four to three. Both figures are consistent with the initial estimate of the scope of the solution. The functionality of the system remains the same. Put another way, the original design has been replaced by a simpler version yet; the public interface to the system has remained unchanged.

5.4.2 Implication Chart

The second approach that we’ll discuss is called the *implication chart* algorithm. We’ll use the same problem that was used for the *row matching* algorithm.

The implication chart algorithm says,

1. Construct the implication chart.
2. There is one square for each pair of states.
3. For each square labeled $S_i, S_j$.
   - If system’s outputs for those states differ enter an “X” in that square.
   - Otherwise write down implied state pairs that must also be merged if $S_i$ and $S_j$ are merged.
4. Advance through chart top-to-bottom and left-to-right.
5. If square $S_i, S_j$ contains an implied merger between state pair $S_m, S_n$ and that pair labels a square already labeled "X".
6. Then $S_i, S_j$ is labeled "X" – the implied merger is not allowed.
7. Continue until no new squares are marked with "X".
8. For each remaining unmarked square $S_i, S_j$, these states are equivalent and be merged.

### Step 1

For the parity generation system, from the states identified in the state table, the *implication chart* in Figure 5.36 elaborates all of the present state pairs.

![Implication Chart for a Sixteen State FSM](image)
Step 2a

The reduction begins by first marking all state pairs with incompatible outputs. That is, for example, one output set may require the output to be \{0 0\} for the two different values of the input data bit as in the case of state 0 and another output set may require the output to be \{1 1\} as in the case of state 8. Clearly the output cannot satisfy both output values simultaneously, thus, state 0 and state 8 cannot be merged. Their behavior is not equivalent for the same inputs.

After completing step 2, the initial set of incompatible states has been identified. These are now reflected in the implication chart in Figure 5.37.

Step 2b

Next all pairs of states are examined and the implications of merging them are identified and noted. That is, additional states, if any, that must also be merged are identified. For example, consider states 0 and 1. If these two states are merged into a single state, the next state entries demand that states (1 and 3) and (2 and 4) must also be merged. Note is made of this by entering the implication into the square 0-1 – for states 0 and 1. The step is repeated with all remaining combinations to give the following complete implication chart.

The resulting implication chart follows in Figure 5.38.

Step 3

Next, all state pairs are examined and the consequences of the implied mergers are assessed. If any of the implied required mergers has been disallowed, the merger is not permitted. For example, if states (6 and 13) are merged, states (0 and 13) and (0 and 14) must also be merged. The square for (0 and 14) contains an X indicating that such a merger is not allowed; thus, the square for (6 and 13) gets marked with an X.
The process is repeated for all squares and associated implications. The completed implication chart now appears in Figure 5.39.

![Figure 5.39 Final Implication Chart](image)

From the chart, it’s evident that states (4 and 5) can be merged – the square is not marked with an X – but, if merged, then (9 and 11) and (10 and 12) must also be merged. Similarly, states (3 and 6) can be merged, but, if merged, then (7 and 13) and (8 and 14) must also be merged.

Following the mergers, the initial reduced set of states is given as

<table>
<thead>
<tr>
<th>States</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>State 0</td>
</tr>
<tr>
<td>1</td>
<td>State 1</td>
</tr>
<tr>
<td>2</td>
<td>State 2</td>
</tr>
<tr>
<td>(3, 6)</td>
<td>State 3a</td>
</tr>
<tr>
<td>(4, 5)</td>
<td>State 4a</td>
</tr>
<tr>
<td>(7, 13)</td>
<td>State 7a</td>
</tr>
<tr>
<td>(8, 14)</td>
<td>State 8a</td>
</tr>
<tr>
<td>(10, 12)</td>
<td>State 10a</td>
</tr>
<tr>
<td>(9, 11)</td>
<td>State 9a</td>
</tr>
</tbody>
</table>

Observe that the following additional mergers are also possible with no implications.

<table>
<thead>
<tr>
<th>States</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7a, 10a)</td>
<td>State 7b</td>
</tr>
<tr>
<td>(8a, 9a)</td>
<td>State 8b</td>
</tr>
</tbody>
</table>

- 29 -
The final reduced set of states is given as,

<table>
<thead>
<tr>
<th>State 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
</tr>
<tr>
<td>State 2</td>
</tr>
<tr>
<td>State 3a</td>
</tr>
<tr>
<td>State 4a</td>
</tr>
<tr>
<td>State 7b</td>
</tr>
<tr>
<td>State 8b</td>
</tr>
</tbody>
</table>

Which gives the same reduced state table we found using the row matching algorithm.

### 5.5 Summary

In this tutorial, we built on the basic concept of the sequential circuit by including input and output capabilities to the designs we studied previously. We examined the state assignment problem and presented several different schemes for assigning values to state variables. Such schemes included binary, Gray, and One Hot encoding. We studied both RTL and structural level models in Verilog for the FSM. We introduced the row matching and implication chart methods for reducing the complexity sequential machines.
References


