Finite State Machines

Introduction

We’ve looked at fundamental building blocks
    Found in embedded systems designs
Now want to briefly review control of such systems

Control component typically expressed as
    Finite state machine
    Collection of such machines
Such devices form basis for control
    Most modern computing and control systems

Fundamental Mealy and Moore models
    Adequate for
    Introducing FSM concepts
    Expressing and implementing small designs

Expressive power limited for larger systems
    Combinational explosion
    When trying to develop input equations
    Quickly limits utility

Basic concepts however do carry forward
    Have wide application

Will first do quick review of basic concepts
Then explore how we can put them to work

Implementation of FSM may be in hardware or software

Hardware implementation of such machines
  • LSI
  • Arrayed logic
  • PLD
  • ROM
  • Discrete logic

Earlier in our studies
    We’ve looked at the basic storage element
    Some simple counting and dividing circuits
Simple state machines
Like counter, divider, timer
Have
   No inputs other than clock
   Only primitive outputs

Such machines referred to as *autonomous clock*
We have classed such basic machines as part of datapath
   In some applications can also be considered as part of control

As we move to more complex designs
   Introduce
      Inputs
      Outputs offering rich functionality
         Such functionality based upon
         State of the machine
         Inputs to the machine

Our high-level block diagram begins with following

Now we have
   Set of inputs
   Set of outputs
      Important to recognize
         Outputs may be
            State variables
            Combinations of state variables
            Combinations of
            State variables
            Inputs

Let’s increase the level of detail of our state machine
   We’ll reflect the
      Inputs
      State variables
      Outputs

We see that our state variables
   Fed back as inputs to our system

We’re now looking at the essence of the strength of the machine
   It has the ability to
      Recognize the state that it is in
      Based upon the values of the state variables
      React based upon that information
      Decision as to which state to go to next now based upon
The current input
The state that the machine is currently in

Let’s continue increasing the level of detail
We’ll increase our view to now include
Storage elements comprising the machine
Combinational logic
  Implements output functionality
  Input equations to storage elements

Our block diagram now becomes

We now see that we have
  n inputs
  m outputs
  p state variables

Associated with each state variable
  We have a memory device
  At this point we do not specify the particular type

Working from this model
  We can begin to formalize out model of the finite state machine

Our model must reflect
  Inputs
  Outputs
    Which may be a function of
      Inputs and State variables
      State variables alone
  State variables
  Movement between states

**Finite State Model**

*Finite State Machine*

  Also known as finite automaton
  Abstract model describing
    Sequential machine

  Forms basis for understanding and developing
    Various computational structures
We now formally define such a finite state machine

We specify the variables

$X_i$ - Represent system n inputs
$Z_j$ - Represent system m outputs
$Y_k$ - Represent internal p state variables

We define our finite state machine as a quintuple

$$M = \{ I, O, S, \delta, \lambda \}$$

$I$ - 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

$x$ is the Cartesian or cross product
The Cartesian product of two vectors
Gives matrix of all possible pairs
Among elements of two vectors

To reflect the different ways of expressing our output
We define

Mealy machine - $\lambda_1$
Output function of
Present state and inputs

Moore machine - $\lambda_2$
Output function of
Present state only

Putting State Machines to Work
Let’s take a detailed look at how we can begin to use our model

We’ll begin with a simple pattern or sequence detector
Couple of possible uses for such a system
Communications systems
Synchronize or lock onto incoming data stream
Based upon initial or synchronizing pattern

CD or DVD player
Similarly using synchronizing pattern to
Sync local timing system to data being read from device
Permit accurate sampling
We will accept data coming into our system
  Data will come in serially
    One bit at a time
  Our specifications require
    We detect the pattern 1010 in the data
    If we recognize the pattern
      We are to output a found signal

The immediate implication of specification
  Time is an important component of our design

Let’s start to analyze the problem
  High-level system requirements
    Input
      One input
        Serial stream of data
    Output
      One output
        Logical 1 every time sequence 1010 detected
  As a first step
    We try to capture the requirements in a block diagram
      Will begin at very high level
      Intent is to capture most important details
        Abstract remainder away

For this problem we have the following simple diagram

Two alternate fundamental algorithms apply to such detection problems

For n bit pattern
  1. Use n bit block window
      collect n bits as working set
    repeat
      if working set matches pattern
        succeed
      else
        build new working set by moving window n bit positions to left
      until done
2. Use n bit sliding window
   collect n bits as working set
   repeat
     if working set matches pattern
       succeed
     else
       build new working set by moving window 1 bit position to left
   until done

Next we formally express the behaviour of the system
   Capture such behaviour in state diagram
   Such a diagram
     Expresses the behaviour of the system in time
     Identifies each legal state of the system
     Depicts means by which system got to each state

**State Diagram**
   State diagram derives from field of mathematics called Graph Theory
   State diagram and graph equivalent

   Basic components of graph
     Vertices
     Arcs

   **Vertices**
     Each state in machine
     Corresponds vertex in state diagram
     Each vertex in the diagram
     Corresponds to state in system
     If the system has 10 states
     We ill have 10 vertices

   **Arcs**
     Arcs interconnect Vertices
     Arcs may be
       Undirected
         Have no specific orientation
         Undirected graph
       Directed
         Have specific orientation
         Directed graph
Cyclic and Acyclic Graphs

Many classes of graph

Two important classes

Cyclic and acyclic

Cyclic
- Can have closed path or cycle
  - From one vertex
  - Through several other vertices
  - Back to original vertex
- Commonly found in flow of control applications

Acyclic
- Does not have closed path or cycle
- Unidirectional path(s) through graph

Building a Graph

Work with directed cyclic graph

Set of vertices to represent states
- Will have finite number of states – Finite State Machine - FSM
  - Finite number of vertices
Set of arcs to represent transitions between states
  - Finite number of arcs

For each vertex
- We have
  - \(2^n\) directed arcs
    - One for each combination of inputs
  - Arcs correspond to state transitions
    - Caused by input variables

These arcs express the mapping we saw earlier
- \(I \times S \rightarrow S\)

If we have 3 inputs to system
- Each state will have \(2^3\) arcs leaving
  - One arc for each possible input combination

This is where Mealy and Moore machines
- Begin to run into trouble
- Many systems – even if reduced
  - Have many more than 4 inputs
- A system with 6 inputs would have
  - 64 arcs leaving each state
- Independent of whether such transitions are meaningful
For Mealy machine
Each directed arc
Labeled with
Input combination
Caused transition
Resulting output symbol that is generated
Associated with each arc we write
input / output

For Moore machine
Directed arcs
Unlabeled
Within circle representing each state
Identify output set for that state
Illustrated in accompanying diagram
State with 5 outputs

Behaviour
Graph or State diagram describes our system
It shows
Succession of states – vertices
Through which sequential machine passes
Output sequence which it produces
Path(s) – arcs
Expressing succession of states
Interconnecting states

In many systems we can define two distinguished states
Initial State
State of machine prior to application of input sequence

Final State
State of machine after application of input sequence

Let’s see what the state diagram for our system will look like
Before we can begin
Need to clarify the specification
Consider that we receive the sequence ….10101010….

As noted above we have two possible interpretations
1. We examine the pattern in groups of 4
   If we detect the pattern
     We must start over looking for the pattern again
   For the above sequence
     We will detect the pattern twice
2. We can reuse part of the pattern through a sliding window
   For the above pattern
   We will detect the pattern 3 times

   We will build the second interpretation as a Mealy machine

   We begin in the *idle* state
   Most of our designs use an initial or idle state
   We will label that state as state A

   Since we only have a single input
   We will only have two arcs emanating from each state

   From the idle state – we’re at t₀
   Our input can be either a 0 or a 1
   If we get a 0
     We will go to state B
     Says that we now have 1 bit correct
   If we get a 1
     We remain in state A
     Says that we have no bits correct
   Our state diagram becomes

   Observe that we label each arc
   Showing input and output combination

   From state B - we’re at t₁
   Once again our input can be either a 0 or a 1
   If we get a 0
     We remain in state B
     Says that we still only have 1 bit correct
   If we get a 1
     We go to state C
     Says that we have two bits correct
   Our state diagram becomes

   Observe that our output remains a 0
   It must since we have not detected the full sequence yet
From state C – we’re at t₂
   Once again our input can be either a 0 or a 1
   If we get a 0
       We’re winning – we go to state C
       Says that we now have 3 bits correct
   If we get a 1
       We fail and must return to A
       Says that we have no bits correct

Our state diagram becomes

Observe that our output remains a 0
   It must since we have not detected the full sequence yet

Now we’re looking for the 4ᵗʰ bit
From state D – we’re at t₃
   Yet again our input can be either a 0 or a 1
   If we get a 0
       We’re only partially win – we go to state B
       Says that we now have just 2 bits correct
   If we get a 1
       We return to C and output a 1
       Says that
           We have matched the patterns
           Also we still have three bits correct

Our final state diagram becomes

We have now captured in graphical presentation
   Complete behaviour of our system

We can use such a diagram for discussion and analysis
   Need somewhat different format
       To begin next stage of design

The State Table
   The state table contains exactly same information as state diagram
     Expresses in tabular form
     Easier to develop logic equations from
   Let’s take a look
A state table has
Two major subdivisions
One column
Showing the state of the system at time $t_n$
A set of columns
Showing the state of the system at time $t_{n+1}$
Each column reflects the next state for each specific input combination

Thus in general we have
$p$ columns
One for each combination of input symbols in set $I$
n rows
One or each state in $S$

This is now the matrix we referred to earlier
The columns give us the input combinations
The rows give us the set of present and next states

To implement machine will require
$k$ memory elements
$k$ is smallest integer value such that
$k \geq \log_2 n$

For our design
We will have two columns
One for the input data taking a value of logical 0
One for the input data taking a value of logical 1
We will have 4 rows
One for each state

Taking the information directly from the state diagram we have

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

The Output Table
The next step in our process is to specify the outputs from our system
These appear in an *output table*

Once again this information
Captured directly from the state diagram
Often we will combine the state and output tables into single table.

The output table has
- \(p\) columns
  - One for each combination of input symbols in set \(I\)
- \(n\) rows
  - One for each state in \(S\)

The output table gives us our output matrix.

Here we have
- \(I \times S \rightarrow O\)
- \(S \rightarrow O\)

For each combination of input symbol and present state:
- Specifies output of system
- Remember
  - If building a Moore machine
    - Each output column will have same value

Once again reading directly from the state diagram:
- The output table for our design is given as:

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

**State Assignment**

Our next step is to
- Choose state variables
- Assign appropriate combinations to each state
  - Let’s us uniquely identify each state in our system

Selecting state assignment
- Important step in design of finite state machine

There are wide variety of techniques of varying degrees of complexity

Let’s look at four

1. **Binary Assignment**
   - Easiest is to simply use binary sequence
     - Initial state gets pattern of binary 0
       - Typically 0 chosen as value for initial state
       - At power on or when necessary
         - Storage elements in system reset by master reset signal
Reset forces output of elements to 0
Thus 0 is natural state

Each subsequent state gets next binary number

Often works with no problem
One major limitation
Building outputs as combinational patterns
Because of hazards and races associated with combinational logic

With such an approach
Run risk of having decoding spikes on our outputs
If output signals utilized to
Clock, strobe, gate, or enable
Other devices
Can have significant problems

2. Gray Assignment
One method to address problem of race conditions and hazards
Ensure that we have only single variable change between states
Reduces chances of decoding spikes

As first step
We develop table listing all states

For each state
Identify
Preceding states
Next states
For our system we have

<table>
<thead>
<tr>
<th>Previous</th>
<th>Present</th>
<th>Next</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>

Then we assign states such that
Single variable change
Previous to present
Present to next
Not always able to do
Using a Karnaugh map helps

For our system
We have 4 states
Requires we use 2 state variables
Thus we now have

<table>
<thead>
<tr>
<th>M</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

Our state assignment now becomes

<table>
<thead>
<tr>
<th>M N</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 0</td>
</tr>
<tr>
<td>B 0</td>
</tr>
<tr>
<td>C 1</td>
</tr>
<tr>
<td>D 1</td>
</tr>
</tbody>
</table>

Generally we select our initial state to have the value of binary 0
Simplifies ensuring system starts in known state

Not always possible to ensure that all adjacencies satisfied
Under such circumstances
Work to satisfy necessary ones

Based upon implementation
Implication on next two schemes
Designs architected as Moore machines
Outputs function of state only

3. One Hot Code
Another method for addressing the problem of
Combinational logic hazards on output signals
Simply avoids them
With One Hot state encoding trade off
Reduced combinational logic for additional storage elements
One Hot implies that bit pattern assigned to any state
Has at most a single one and the remainder all set to 0

Of question is the initial state
If master reset sets all storage devices to 0 state
Initial state will be all 0’s
Otherwise
Initial state will be assigned pattern such as
…000001
For our four state machine possible assignments might be

<table>
<thead>
<tr>
<th>LMN</th>
<th>LMNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 000</td>
<td>A 0001</td>
</tr>
<tr>
<td>B 001</td>
<td>B 0010</td>
</tr>
<tr>
<td>C 010</td>
<td>C 0100</td>
</tr>
<tr>
<td>D 100</td>
<td>D 1000</td>
</tr>
</tbody>
</table>

Obvious weakness of One Hot Code
For large number of states
Number of storage devices grows quite quickly

4. M of N Encoding
M of N encoding scheme
Simple variant on One Hot encoding
Rather than permitting only single one in state assignment pattern
Any M of the possible N state variables may be set to 1
Key aspect of approach
Each state variable corresponds to an output variable
Since output derives directly from state variable storage element
Cannot possibly have race conditions and hazards
Will look at example of how this might apply shortly

The Transition Table
Once we have the state assignment

We use it to construct a *transition table*

The transition table combines our state table
With the gray state assignment we just developed

We simply substitute the state variable combination for each state
Back into the state table

For our design we have

<table>
<thead>
<tr>
<th>PQ</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = 0</td>
</tr>
<tr>
<td>A 00</td>
<td>B 01</td>
</tr>
<tr>
<td>B 01</td>
<td>B 01</td>
</tr>
<tr>
<td>C 11</td>
<td>D 10</td>
</tr>
<tr>
<td>D 10</td>
<td>B 01</td>
</tr>
</tbody>
</table>
The Input Equations
We’re now ready to implement the design
For the state equations
Simply follow same procedure we did earlier

For our design
We first develop the following K Maps

<table>
<thead>
<tr>
<th>PQ</th>
<th>0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>1</td>
<td>P</td>
</tr>
</tbody>
</table>

If we choose to use D flip flops
We write the following equations from the maps

\[ D_p = \overline{P}QX + PQ\overline{X} + P\overline{Q}X \]
\[ D_q = \overline{P}\overline{X} + \overline{P}Q + P\overline{Q} \]

The Output Equations
The output equations follow in a similar manner
We simply use a K Map

For our design we have

<table>
<thead>
<tr>
<th>PQ</th>
<th>0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>0</td>
<td>Out</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>1</td>
<td>Out</td>
</tr>
</tbody>
</table>

\[ Out = P\overline{Q}X \]

Example
System must output an 8 bit data word in two 4 bit pieces
Least significant nibble then most significant nibble
According to the following sequence.

When a \textit{Ready} signal is received,
1. Generate a signal \textit{E1} to output the lower 4 bits.
2. Wait \( \tau_d \) then output the signal \textit{dStrobe}.
3. Wait \( 2\tau_d \) then terminate the signals \textit{E1} and \textit{dStrobe}.
4. Generate a signal \textit{E2} to output the upper 4 bits.
5. Wait \( \tau_d \) then output the signal \textit{dStrobe}.
6. Wait \( 2\tau_d \) then terminate the signals \textit{E2} and \textit{dStrobe}.
System has
Six states
S0..S5
Four inputs
POR, Ready, τd, 2τd
Four outputs
E1, E2, dStrobe, countEnab
Design assumes counter to time the two delay intervals

Using M of N encoding
Can specify state variable values

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

For this design state variables double as following outputs
L – countEnab
M – E1
N – E2
P – dStrobe

Design Guidelines
From our work we propose the following guidelines
1. From word description of 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 transition and output tables
6. Develop Karnaugh map for each state variable
7. Select memory device and develop input equations from Karnaugh map
8. Develop Karnaugh map for each output variable
9. Develop output equations from Karnaugh map

Multiple Machines
When designing complex systems
Good practice to decompose into smaller pieces
We do this when
Working with object centered language
Executing functional decomposition of system
Developing hw or sw modules
Using such a scheme
Utilize slave machines
To implement individual pieces of functionality
Master determines when each slave is enabled
Based upon aspect of problem being executed

Consider system with 4 major tasks
Each task comprised of multiple simpler tasks

System now consists of 5 sequential machines
Master controller
4 Slave controllers

Architecture appears as illustrated in adjacent drawing

Control of slaves can be implemented in several ways
• All devices respond to main system reset
  Enable signal going from master to each slave
• Main system reset to master
  Reset control to each slave derives from disjunction of
    Master reset
    Secondary reset derived from master
  Advantage
    Devices held in reset state until needed

Either implementation could hold slave devices
Unpowered or in low power mode until necessary