10 Finite state machines (FSMs)
- Synchronous sequential circuits (FSMs)
- Moore and Mealy types FSMs
- State diagram
- State table
- State assignments and one-hot encoding
- FSM Designing procedure
- Example: a BTS 15-baht ticket dispenser
- Verilog for FSMs
- **Testing your design**

Synchronous sequential circuits
- Completing the digital system “loop”

![Diagram](https://example.com/diagram.png)

- clock regulates speed of digital system
Synchronous sequential circuits
- Eliminating input and output managers

Importance of finite state machines (FSMs):

**ANY SYNCHRONOUS CIRCUIT**
(i.e., what we do in this course)
**IS A FINITE STATE MACHINE OF SOME FORM.**

This means:
- Combinational logic is a FSM w/o memory
- Flip-Flops and Counters are also FSMs
Synchronous sequential circuits
- All digital data, running off single clock

What is in the “compute” block?
- Only combinational circuits

Custom designed circuits from
1. Boolean equations – using K-map or other methods
2. MSI components – adders, subtractors, multiplexers, decoders, encoders, comparators, etc.

Basically, anything that does not have any memory.

So, \[ \{ D, \text{output} \} = \text{function (input, Q)} \]
What is in the “store” block?

- Only flip-flops, so Q is D delayed by clock.
- **No computation.** Put all computation in “compute” block.

\[
q \leftarrow d \text{ after (usually) posedge clk}
\]

---

New names for D and Q

- D and Q are usually called *state*.
- The complete circuit with compute and store block is called a finite state machine (FSM) or synchronous sequential circuit.

- FSM is very powerful
- Almost every computer today can be modeled as a collection of FSMs.
- Designing FSMs in Verilog (or VHDL) is *relatively* easy.
Moore model for FSMs:

Edward Moore’s model: MOORE MODEL
\[ D = F(W, Q) \]
\[ Z = G(Q) \]

where F and G are some Boolean functions

Output depends only on current state (Q)
Input (W) and current state (Q) determine next state (D)

Mealy model for FSMs:

George Mealy’s model: MEALY MODEL
\[ D = F(W, Q) \]
\[ Z = G(W, Q) \]

where F and G are some Boolean functions

Output depends only both current state (Q) and input (W)
Input (W) and current state (Q) determine next state (D)
Moore machine design example
- Design a simple 11 sequence recognizer

1. Input is 1-bit wide, called w, output is also 1-bit wide, called z.
2. Output z is 1 if w during the past two clock cycles are 1, otherwise z is 0.

Sample sequence:

<table>
<thead>
<tr>
<th>Clockcycle:</th>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
<th>t₅</th>
<th>t₆</th>
<th>t₇</th>
<th>t₈</th>
<th>t₉</th>
<th>t₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>w:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>z:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

State diagram
Graph that represents what happens

1. Start out with an initial state that FSM will be in after reset.
2. We can name that state A for now (more later)
3. Tag the output for this particular state
   → Since the FSM just starts out, we assume the worst:
      It saw two 0’s in the past two clock cycle, so the output must be zero (z=0)

The circle is called a node or a state.
State diagram

List all possible inputs from state A

- Two possible inputs, w=0, and w=1
  * We must list EVERY possible input combinations.
  - If w=0, then it’s the same as if we started out from scratch
  - If w=1, then we already receive one 1
  - So, state A = Seen 0 in the last clock
    state B = Seen 01 – output must still be zero, since
    we only saw one 1

The arrowed line is called an arc or a state transition.

State diagram

List all arcs from state C

- Now state C is “seen 11” so, output from state C
  must be 1
- We also must list all arcs from state C, too
State diagram

List all arcs from state C

We now have a closure, meaning that all arcs no longer lead to new states → State diagram is complete.

State table

Similar to state diagram, but in table format:

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>w = 0</td>
<td>w = 1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Note: This is figure 8.3 from text. Note that reset state info is missing from the table!

One problem remains: digital systems have only 0 & 1. We have to convert A, B, C to some binary numbers. → This is called state encoding or state assignment.
State assignment

3 states, so 2 bits of information would be sufficient (2 bits = 4 possible codes)

Let’s assign A=00, B=01, C=10, and leave 11 unused. Rewrite state table:

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output z</th>
</tr>
</thead>
<tbody>
<tr>
<td>w = 0</td>
<td>w = 1</td>
<td></td>
</tr>
<tr>
<td>y_2 y_1</td>
<td>y_2 y_1</td>
<td></td>
</tr>
<tr>
<td>A 00</td>
<td>00 01</td>
<td>0</td>
</tr>
<tr>
<td>B 01</td>
<td>00 10</td>
<td>0</td>
</tr>
<tr>
<td>C 10</td>
<td>00 10</td>
<td>1</td>
</tr>
<tr>
<td>11 dd</td>
<td>dd</td>
<td>d</td>
</tr>
</tbody>
</table>

See that y_1 & y_2 become present state variables.

Y_1 & Y_2 are next state variables.

Together they are state variables.

Sequential circuits for Moore FSM

1. Next state (Y) will become present state (y) after a clock “tick” (clock tick = posedge of clock)
2. Output depends only on current (or present) state
Let’s design \( F \) and \( G \) from state table:

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w = 0 )</td>
<td>( w = 1 )</td>
<td></td>
</tr>
<tr>
<td>( 00 )</td>
<td>( 00 )</td>
<td>0</td>
</tr>
<tr>
<td>( 01 )</td>
<td>( 00 )</td>
<td>0</td>
</tr>
<tr>
<td>( 10 )</td>
<td>( 00 )</td>
<td>1</td>
</tr>
<tr>
<td>( 11 )</td>
<td>( dd )</td>
<td>( d )</td>
</tr>
</tbody>
</table>

\( Y_1 = F_1 \) (current state, input) = \( F_1 (y_1, y_2, w) \)
\( Y_2 = F_1 \) (current state, input) = \( F_2 (y_1, y_2, w) \)

next state is derived from current state and input

\( z = G \) (current state) = \( G (y_1, y_2) \)

output is derived from current state

Design \( F_1, F_2, \) and \( G \) using K-map:

\( Y_1 = w \bar{y}_1 \bar{y}_2 \)
\( Y_2 = w y_1 \bar{y}_2 + w \bar{y}_1 y_2 \)
\( z = \bar{y}_1 y_2 \)
Some notes

The design takes advantage of Q’ output from flops.

Note that in this (figure 8.8) reset is provided asynchronously.

Timing diagram of finished circuit:

Changes in y1, y2, and z occur after posedge clk, because they are gated by D-flops that implement the states.
Mealy machine design example
Still a simple 11 sequence recognizer

1. Input is 1-bit wide, called w, output is also 1-bit wide, called z.
2. Output z is 1 if w in the clock cycles is 1, and the current input (w) is 1, otherwise z is 0.

Sample sequence:

<table>
<thead>
<tr>
<th>Clock cycle:</th>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
<th>t_8</th>
<th>t_9</th>
<th>t_{10}</th>
</tr>
</thead>
<tbody>
<tr>
<td>w:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>z:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

State diagram for Mealy machine:
- Note that Moore machine's output depends ONLY on present state, so Output of Moore machine is associated with the states (circles)

Mealy output depends on both present state and present input, so

Output of Mealy machine is associated with the arcs (state transitions)
State diagram for Mealy machine:

State tables will be slightly different from Moore’s as well.

State table for Mealy machine:

Output depends on present input \((w)\) as well as state. Since there are only two states, encode \(A=0\), and \(B=1\), so

Find \(Y = F (y, w)\) and find \(z = G (y, w)\) like before, and…
Circuit for Mealy machine example:

Note that present input can affect present output

Reset is provided

Timing diagram Mealy machine example:

State assignment approaches:
Two common approaches:

1. **Minimal number of bits to encode states:**
   Using at most \( \lceil \log_2(\# \text{ of states}) \rceil \)

2. **One-hot encoding:** n-bit for n-state FSM
   Encode states like this:
   - A = 00001
   - B = 00010
   - C = 00100
   - ...
FSM design yesterday (pre-HDL):
   Can be extremely tedious if done manually

FSM design today:
   Most newer designs are done via HDL, like Verilog. For designing FSM in Verilog, you can use minimal number of bits encoding, and let the synthesis tool optimize the logic.

FSM design procedure:
1. Write clear functional description (what you want the FSM to do)
2. Design state diagram
   a) start from initial state
   b) add arcs and more states until closure
   **c) make sure every input combination is listed for every state.
>3. Turn state diagram into state table, minimize, and implement circuits
>3. Code Verilog from your state diagram
Where FSM design can be hard:
1. Writing incomplete functional description
   → Think of all corner cases carefully

2. Writing incomplete state diagram
   → Make sure you have the arcs for every input combination (3-bit input will have 8 arcs out from every state)

3. State diagram does not have closure
   → Try to find existing state where the new arcs can point to.

Example: a 15-baht BTS ticket dispenser:
Bangkok Skytrain ticket machine sells 15-baht tickets.
- User can put in 5-baht or 10-baht coin, one at a time only.
- Clock is 1MHz (quick enough so user cannot put coin in every clock cycle). explain why needed for vending FSM
- The ticket costs 15 baht, and machine may need to give 5 baht change if user put in more than 15-baht.
Example: a 15-baht BTS ticket dispenser:
- User can put in 5-baht or 10-baht coin, one at a time only.
- This means that \{b,a\} will never be 11.**designer’s spec**
- This also means that we will never need to account for more than 20-baht (You’ll see why later)

Memorize the signals (a, b, c, t) and let’s do a **Moore FSM**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No coin in</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5-baht in</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10-baht in</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10-baht in</td>
</tr>
</tbody>
</table>

Since it’s Moore machine, only input will be associated with every arc, so we will write this:

```
ba
```
Some characteristic of the inputs:
Since clock is quick enough that user cannot put in coin for every cycle, \{b,a\} must be 00 every other cycle or more. (Suppose clock = 1 MHz (extremely slow), can you still put in two coins within 1 microsecond of each other?)

Example: a 15-baht BTS ticket dispenser:
Name the states after the amount of money in the machine:
Add reset: should reset to zero baht (explain why)

Zero state now has all the arcs for every input 00,01,10,11

No coin in, 5-baht in, and 10-baht in – no ticket or change
Do the same thing for “five” state

Do the same thing for “ten” state
For “fifteen,” dispense ticket and go to “zero”- no change

Because to arrive at “fifteen” a coin must be put in, and no coin comes in during the next clock, letting us reset safely

our FSM will be unfair to users who can put in two coins within 1us of each other – they are X-Men and must be punished
Same with “twenty,” dispense ticket and give 5-baht back

Here’s the \{t,c\} output associated with each state:

We have not done state encoding yet!
State encoding for the ticket dispenser:
- 5 states, so \( \text{CEIL}[\log_2(5)] = 3 \rightarrow 3 \) bits
- Let’s just iterate

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>3’b000</td>
</tr>
<tr>
<td>five</td>
<td>3’b001</td>
</tr>
<tr>
<td>ten</td>
<td>3’b010</td>
</tr>
<tr>
<td>fifteen</td>
<td>3’b011</td>
</tr>
<tr>
<td>twenty</td>
<td>3’b100</td>
</tr>
</tbody>
</table>

Every synchronous digital systems **can** be modeled as a finite state machine.

**BUT**

**Not** every synchronous digital system **should** be modeled as a finite state machine.

An example of “should not”: a 32-bit counter

\( \rightarrow \) More than 4 billion states
Verilog for FSMs:

If you still don’t understand the following after the lecture, you will have a big problem with this course, especially on exams. See the teaching staff for help before it is too late.

Verilog for FSMs:
Idea: Just copy hardware:

Moore machine:

output = G (present state)
Mealy machine:

output = G (present state, input)

Each “box” is then a behavioral code

Recommended coding style:
- Name “present state” just “state”
- Name “next state” as “next_state”

And also for this example:
- Name “input” as “in”
- Name “output” as “out”
- Name “clock” as “clk”
Next state combinational logic box:

always @ (in or state) begin
    // compute next_state
    next_state = f(in, state);
end

State memory box:

always @ (posedge clk) begin
    if (reset)
        state <= #1 RESET_VALUE;
    else
        state <= #1 next_state;
end
Output combinational logic box (Moore):

```verbatim
class always @ (state) begin
  // compute output
  out = g(state);
end
```

Output combinational logic box (Mealy):

```verbatim
class always @ (in or state) begin
  // compute output
  out = g(in, state);
end
```
module fsm (in, out, clk, reset);
  // port declaration here
  always @(in or state) begin
    next_state = f(in, state);
  end
  always @(posedge clk) begin
    if (reset)
      state <= #1 RESET_VALUE;
    else
      state <= #1 next_state;
  end
  always @(in or state) begin
    out = g(in, state);
  end
endmodule

The spirit of Verilog design:

- Don’t try to write long pieces of code like a software program

- Break complex problem into small pieces

- Hook up all the pieces together in a logical way, using block diagrams to help if necessary.
Verilog code for our ticket machine 1:

Text macro definitions, module, and port declarations

```
`timescale 1ns/1ns
`define clk2q #10

`define zero 3'b000
`define five 3'b001
`define ten 3'b010
`define fifteen 3'b011
`define twenty 3'b100
```

Module bts_fsm (a, b, t, c, clk, reset);

input a, b, clk, reset;
output t, c;
reg t, c;
reg [2:0] next_state, state;

Verilog code for our ticket machine 2:

Behavioral code for “next state” logic.

```
always @(a or b or state) begin
    case (state)
        `zero: case ((b,a))
            2'b00: next_state = `zero;
            2'b01: next_state = `five;
            default: next_state = `ten;
        endcase
        `five: case ((b,a))
            2'b00: next_state = `five;
            2'b01: next_state = `ten;
            default: next_state = `fifteen;
        endcase
        `ten: case ((b,a))
            2'b00: next_state = `ten;
            2'b01: next_state = `fifteen;
            default: next_state = `twenty;
        endcase
        default: next_state = `zero;
    endcase
end
```
Verilog code for our ticket machine 3:
Memory and behavioral code for “output” logic.

```
// state memory behavioral block
always @ (posedge clk) begin
    if (reset)
        state <= `clk2q `zero;
    else
        state <= `clk2q next_state;
end

// output logic behavioral block
always @ (state) begin
    case (state)
        `fifteen: {t,c} = 2'b10;
        `twenty: {t,c} = 2'b11;
        default: {t,c} = 2'b00;
    endcase
end
endmodule
```

Synthesis report for our code:
The synthesis tool is very impressive indeed.

```
Synthesizing Unit <bts_fsm>.
Related source file is bts_fsm.v.
Found finite state machine <FSM_0> for signal <state>.
-----------------------------------------------------------
| States             | 5 |
| Transitions        | 11 |
| Inputs             | 2 |
| Outputs            | 2 |
| Reset type         | synchronous |
| Encoding           | automatic |
| State register     | d flip-flops |
-----------------------------------------------------------
Summary: inferred 1 Finite State Machine(s).
Unit <bts_fsm> synthesized.
- Found all 5 states
- Found all 11 arcs
- Found 2 inputs (a, b)
- Found 2 outputs (t, c)
- Found that we used D-flops for our design
```
**Simulation result for our code:**

Let’s see what happens with this design is in use.

![Simulation Diagram]

Always reset the FSM before use.

Three 5-baht coins in, ticket out, no change.

Two 10-baht coins in, ticket out, change.

One 5-baht followed by one 10-baht coin, ticket out, no change.

**Testing of FSMs:**

**Question:** We have an FSM design, how do we know that it is working as designed?

**Answer:** Smart, efficient testing can eliminate most bugs.

- Test every arc
- Test \{prev\_arc, present\_arc\} matrix
  extend the idea to
- Test \{prev\_prev\_arc, prev\_arc, present\_arc\}
  then extend it further
Testing every arc:

Idea: Make sure that the input makes the FSM under test goes through every transition (arc).

First: Label all the arcs: A, B, C, ..., K (11 arcs)

Second: Make sure your stimulus lets FSM goes through every arc. CHECK OUTPUT

Third: Check that all arcs are traversed and outputs are correct!


→ all arcs are covered.
Testing every arc: A table can help

<table>
<thead>
<tr>
<th>arcs</th>
<th>traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
</tr>
<tr>
<td>E</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>G</td>
<td>Yes</td>
</tr>
<tr>
<td>H</td>
<td>Yes</td>
</tr>
<tr>
<td>I</td>
<td>Yes</td>
</tr>
<tr>
<td>J</td>
<td>Yes</td>
</tr>
<tr>
<td>K</td>
<td>Yes</td>
</tr>
<tr>
<td>R</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>output correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>Yes</td>
</tr>
<tr>
<td>five</td>
<td>Yes</td>
</tr>
<tr>
<td>ten</td>
<td>Yes</td>
</tr>
<tr>
<td>fifteen</td>
<td>Yes</td>
</tr>
<tr>
<td>twenty</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Moore machine: 2 tables: one to check all arcs, the other to check whether the output of every state is correct.

Testing every arc:

<table>
<thead>
<tr>
<th>arcs</th>
<th>traversed?</th>
<th>output correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Z</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Mealy machine FSM testing table

Mealy machine: 1 table: since each output is associated with each arc, you check that the arc is traversed and the corresponding output of that arc is also correct.
Is this good enough for testing?

General answer:

YOU CAN NEVER TEST ENOUGH
YOU CAN NEVER TEST ENOUGH
YOU CAN NEVER TEST ENOUGH

Well, how about a compromise?

How about a test matrix with entries in terms of \{previous\_arc, present\_arc\}?
Example 1: If the previous arc was K, the present arc can only be A, B, C, or R (You can always reset your design any time)

Example 2: If the previous arc was D, the present arc can only be D, E, F, or R.

You can always reset your design.
Testing \{\text{previous\_arc, present\_arc}\}:

How good was our testing in this metric?

The \{\text{previous\_arc, present\_arc}\} pairs that are covered:
RA, AB, BD, DE, EG, GH, HJ, JA, AA, … and so on...

Let’s mark these with “O” in our matrix:

Now you can see that we have not yet tested every possible combination of (prev\_arc, present\_arc)
For example, (D,D) was not tested, and it is possible to test it.
Testing \{\text{previous\_arc, present\_arc}\}:

\[
\begin{array}{cccccccccc}
A & B & C & D & E & F & G & H & I & J & K & R \\
\hline
A & O & O & O & & & & & & & & X \\
B & & O & X & X & & & & & & & \\
C & & & O & X & X & & & & & & X \\
D & & & X & O & O & & & & & & X \\
E & & & & O & X & X & & & & & X \\
F & & & & & O & X & & & & & X \\
G & & & & & X & O & O & & & & X \\
H & & & & & & O & X & & & & X \\
I & & & & & & & O & X & & & \\
J & & O & X & X & & & & & & & X \\
K & & O & X & X & & & & & & & X \\
R & & O & X & X & & & & & & & X \\
\end{array}
\]

\(\{B,E\}\) and \(\{B,F\}\) are not tested because it would require two coins in consecutive clocks.

It is possible to test it in simulation, though.

\[
\begin{array}{cccccccccc}
A & B & C & D & E & F & G & H & I & J & K & R \\
\hline
A & O & O & O & & & & & & & & X \\
B & & O & X & X & & & & & & & \\
C & & & O & X & X & & & & & & X \\
D & & & X & O & O & & & & & & X \\
E & & & & O & X & X & & & & & X \\
F & & & & & O & X & & & & & X \\
G & & & & & X & O & O & & & & X \\
H & & & & & & O & X & & & & X \\
I & & & & & & & O & X & & & \\
J & & O & X & X & & & & & & & X \\
K & & O & X & X & & & & & & & X \\
R & & O & X & X & & & & & & & X \\
\end{array}
\]

We have not tried to reset the FSM while it is running either.

SEE?

We have not tested in every possible way yet!

A bug can be hidden in X’s.
Testing in general:

**IDEA:** We have a logic design, purely combinational or FSM, how do we know that it is working as designed?

**Answer:** Smart, efficient testing can eliminate most bugs.

**Look at Verilog code:**

- every line of code executed
- every case taken
- every if (true) and else (false) case taken
- every output bit that can toggle has toggled
- random testing