Finite State Machines

- Design methodology for sequential logic
  -- identify distinct states
  -- create state transition diagram
  -- choose state encoding
  -- write combinational Verilog for next-state logic
  -- write combinational Verilog for output signals

- Lots of examples
Finite State Machines

- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized “states” of operation.
- At each clock edge, combinational logic computes outputs and next state as a function of inputs and present state.

![Diagram of Finite State Machines](image)

- Inputs and present state are combined with combinational logic to produce outputs and next state.
- Registers store the current state, which is updated at each clock edge.
Two Types of FSMs

Moore and Mealy FSMs: different output generation

- **Moore FSM:**
  - Inputs: $x_0...x_n$
  - Next state: $S^+$
  - Present state: $S$
  - Outputs: $y_k = f_k(S)$

- **Mealy FSM:**
  - Inputs: $x_0...x_n$
  - Next state: $S^+$
  - Present state: $S$
  - Outputs: $y_k = f_k(S, x_0...x_n)$
Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It’s a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

Whenever input L goes from low to high...

...output P produces a single pulse, one clock period wide.
Step 1: State Transition Diagram

• Block diagram of desired system:

![State Transition Diagram](image)

• **State transition diagram** is a useful FSM representation and design aid:

  - **State 00:** Low input, Waiting for rise \( P = 0 \)
    - Transition to state 01 if \( L = 1 \) at the clock edge.
    - Stay in state 00 if \( L = 0 \) at the clock edge.

  - **State 01:** Edge Detected! \( P = 1 \)
    - Transition to state 11 if \( L = 1 \) at the clock edge.
    - Stay in state 01 if \( L = 0 \) at the clock edge.

  - **State 11:** High input, Waiting for fall \( P = 0 \)
    - Stay in state 11 if \( L = 1 \) at the clock edge.
    - Transition to state 00 if \( L = 0 \) at the clock edge.

  - Binary values of states:
    - L=0
    - L=1

  - Synchronizer
    - D Q
    - CLK

  - Edge Detector
    - Level to Pulse FSM
    - P
Arcs leaving a state are mutually exclusive, i.e., for any combination input values there’s at most one applicable arc.

Arcs leaving a state are collectively exhaustive, i.e., for any combination of input values there’s at least one applicable arc.

So for each state: for any combination of input values there’s exactly one applicable arc.

Often a starting state is specified.

Each state specifies values for all outputs (Moore).
Choosing State Representation

Choice #1: binary encoding

For $N$ states, use $\lceil \log_2 N \rceil$ bits to encode the state with each state represented by a unique combination of the bits. Tradeoffs: most efficient use of state registers, but requires more complicated combinational logic to detect when in a particular state.

Choice #2: “one-hot” encoding

For $N$ states, use $N$ bits to encode the state where the bit corresponding to the current state is 1, all the others 0. Tradeoffs: more state registers, but often much less combinational logic since state decoding is trivial.
Step 2: Logic Derivation

Transition diagram is readily converted to a state transition table (just a truth table)

<table>
<thead>
<tr>
<th>Current State</th>
<th>In</th>
<th>Next State</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$ $S_0$</td>
<td>$L$</td>
<td>$S_1^+$ $S_0^+$</td>
<td>$P$</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>1 1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Combinational logic may be derived using Karnaugh maps
Moore Level-to-Pulse Converter

Moore FSM circuit implementation of level-to-pulse converter:

\[ y_k = f_k(S) \]

\[ S_1^+ = LS_0 \]
\[ S_0^+ = L \]

\[ P = S_1S_0 \]
Design of a Mealy Level-to-Pulse

- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations

1. When $L=1$ and $S=0$, this output is asserted immediately and until the state transition occurs (or $L$ changes).

2. While in state $S=1$ and as long as $L$ remains at 1, this output is asserted until next clock.
Mealy Level-to-Pulse Converter

Mealy FSM circuit implementation of level-to-pulse converter:

- FSM’s state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM’s implicit single-cycle assertion of outputs during state transitions
Moore/Mealy Trade-Offs

• How are they different?
  – Moore: outputs = f( state ) only
  – Mealy outputs = f( state and input )
  – Mealy outputs generally occur one cycle earlier than a Moore:

  **Moore:** delayed assertion of P
  **Mealy:** immediate assertion of P

  • Compared to a Moore FSM, a Mealy FSM might...
    - Be more difficult to conceptualize and design
    - Have fewer states
Example: Intersection Traffic Lights

• Design a controller for the traffic lights at the intersection of two streets - two sets of traffic lights, one for each of the streets.

• Step 1: Draw starting state transition diagram. Just handle the usual green-yellow-red cycle for both streets. How many states? Well, how many different combinations of the two sets of lights are needed?

• Step 2: add support for a walk button and walk lights to your state transition diagram.

• Step 3: add support for a traffic sensor for each of the streets - when the sensor detects traffic the green cycle for that street is extended.

Example to be worked collaboratively on the board...
**GOAL:**
Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. The combination should be **01011**.

**STEPS:**
1. Design lock FSM (block diagram, state transitions)
2. Write Verilog module(s) for FSM
Step 1A: Block Diagram

Clock generator

Clock generator

Button Enter

Button 0

Button 1

fsm_clock

reset

b0_in

lock

unlock

reset

b0

state

b1

fsm

button

button

button

Unlock LED

Unlock LED

LED DISPLAY

6.111 Fall 2017

Lecture 6
Step 1B: State transition diagram

RESET
Unlock = 0

"0"
Unlock = 0

"01"
Unlock = 0

"01011"
Unlock = 1

"0101"
Unlock = 0

"010"
Unlock = 0

6 states → 3 bits
Step 2: Write Verilog

module lock(input clk, reset_in, b0_in, b1_in, output out);

// synchronize push buttons, convert to pulses

// implement state transition diagram
reg [2:0] state, next_state;
always @(*) begin
  // combinational logic!
  next_state = ???;
end
always @(posedge clk) state <= next_state;

// generate output
assign out = ???;

// debugging?
endmodule
Step 2A: Synchronize buttons

// button
// push button synchronizer and level-to-pulse converter
// OUT goes high for one cycle of CLK whenever IN makes a
// low-to-high transition.

module button(
    input clk,in,
    output out
);
    reg r1,r2,r3;
    always @(posedge clk)
    begin
        r1 <= in; // first reg in synchronizer
        r2 <= r1; // second reg in synchronizer, output is in sync!
        r3 <= r2; // remembers previous state of button
    end

    // rising edge = old value is 0, new value is 1
    assign out = ~r3 & r2;
endmodule
Step 2B: state transition diagram

```verbatim
parameter S_RESET = 0; // state assignments
parameter S_0 = 1;
parameter S_01 = 2;
parameter S_010 = 3;
parameter S_0101 = 4;
parameter S_01011 = 5;

reg [2:0] state, next_state;
always @(*) begin
    // implement state transition diagram
    if (reset) next_state = S_RESET;
    else case (state)
        S_RESET: next_state = b0 ? S_0   : b1 ? S_RESET : state;
        S_0:     next_state = b0 ? S_0   : b1 ? S_01    : state;
        S_01:    next_state = b0 ? S_010 : b1 ? S_RESET : state;
        S_010:   next_state = b0 ? S_010 : b1 ? S_0101  : state;
        S_0101:  next_state = b0 ? S_010 : b1 ? S_01011 : state;
        S_01011: next_state = b0 ? S_010 : b1 ? S_RESET : state;
        default: next_state = S_RESET;  // handle unused states
    endcase
end

always @(posedge clk) state <= next_state;
```
Step 2C: generate output

// it’s a Moore machine!  Output only depends on current state
assign out = (state == S_01011);

Step 2D: debugging?

// hmmm. What would be useful to know?  Current state?
// hex_display on labkit shows 16 four bit values
assign hex_display = {60'b0, 1'b0, state[2:0]};
Step 2: final Verilog implementation

```verilog
module lock(input clk, reset_in, b0_in, b1_in,
            output out, output [3:0] hex_display);

wire reset, b0, b1; // synchronize push buttons, convert to pulses
button b_reset(clk, reset_in, reset);
button b_0(clk, b0_in, b0);
button b_1(clk, b1_in, b1);

parameter S_RESET = 0; parameter S_0 = 1; // state assignments
parameter S_01 = 2; parameter S_010 = 3;
parameter S_0101 = 4; parameter S_01011 = 5;

reg [2:0] state, next_state;
always @(*) begin // implement state transition diagram
  if (reset) next_state = S_RESET;
  else case (state)
    S_RESET: next_state = b0 ? S_0 : b1 ? S_RESET : state;
    S_0: next_state = b0 ? S_0 : b1 ? S_01 : state;
    S_01: next_state = b0 ? S_010 : b1 ? S_RESET : state;
    S_010: next_state = b0 ? S_0 : b1 ? S_0101 : state;
    S_0101: next_state = b0 ? S_010 : b1 ? S_01011 : state;
    S_01011: next_state = b0 ? S_0 : b1 ? S_RESET : state;
    default: next_state = S_RESET; // handle unused states
  endcase
end
always @(posedge clk) state <= next_state;

assign out = (state == S_01011); // assign output: Moore machine
assign hex_display = {1'b0, state}; // debugging
endmodule
```
Real FSM Security System
The 6.111 Vending Machine

- Lab assistants demand a new soda machine for the 6.111 lab. You design the FSM controller.
- All selections are $0.30.
- The machine makes change. (Dimes and nickels only.)
- Inputs: limit 1 per clock
  - Q - quarter inserted
  - D - dime inserted
  - N - nickel inserted
- Outputs: limit 1 per clock
  - DC - dispense can
  - DD - dispense dime
  - DN - dispense nickel
What States are in the System?

- A starting (idle) state:
  - idle

- A state for each possible amount of money captured:
  - got5c
  - got10c
  - got15c
  - ...

- What’s the maximum amount of money captured before purchase?
  - 25 cents (just shy of a purchase) + one quarter (largest coin)
  - got35c
  - got40c
  - got45c
  - got50c
  - ...

- States to dispense change (one per coin dispensed):
  - got45c
  - Dispense Dime
  - Dispense Nickel
Here’s a first cut at the state transition diagram.

See a better way?
So do we.
Don’t go away...
Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

State Reduction

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.
Verilog for the Moore Vender

**FSMs are easy in Verilog.**
**Simply write one of each:**

- **State register**
  (sequential always block)

- **Next-state**
  combinational logic
  (comb. always block with case)

- **Output combinational**
  logic block
  (comb. always block or assign statements)

```verilog
module mooreVender (
    input N, D, Q, clk, reset,
    output DC, DN, DD,
    output reg [3:0] state);

    reg next;

    parameter IDLE = 0;
    parameter GOT_5c = 1; parameter GOT_10c = 2;
    parameter GOT_15c = 3; parameter GOT_20c = 4;
    parameter GOT_25c = 5; parameter GOT_30c = 6;
    parameter GOT_35c = 7; parameter GOT_40c = 8;
    parameter GOT_45c = 9; parameter GOT_50c = 10;
    parameter RETURN_20c = 11; parameter RETURN_15c = 12;
    parameter RETURN_10c = 13; parameter RETURN_5c = 14;

    always @(posedge clk or negedge reset)
    if (!reset) state <= IDLE;
    else state <= next;

endmodule
```
always @ (state or N or D or Q) begin

    case (state)
        IDLE:      if (Q) next = GOT_25c;
                    else if (D) next = GOT_10c;
                    else if (N) next = GOT_5c;
                    else next = IDLE;
    GOT_5c:    if (Q) next = GOT_30c;
                    else if (D) next = GOT_15c;
                    else if (N) next = GOT_10c;
                    else next = GOT_5c;
    GOT_10c:   if (Q) next = GOT_35c;
                    else if (D) next = GOT_20c;
                    else if (N) next = GOT_15c;
                    else next = GOT_10c;
    GOT_15c:   if (Q) next = GOT_40c;
                    else if (D) next = GOT_25c;
                    else if (N) next = GOT_20c;
                    else next = GOT_15c;
    GOT_20c:   if (Q) next = GOT_45c;
                    else if (D) next = GOT_30c;
                    else if (N) next = GOT_25c;
                    else next = GOT_20c;
        GOT_25c:   if (Q) next = GOT_50c;
                    else if (D) next = GOT_35c;
                    else if (N) next = GOT_30c;
                    else next = GOT_25c;
        GOT_30c:   next = IDLE;
        GOT_35c:   next = RETURN_5c;
        GOT_40c:   next = RETURN_10c;
        GOT_45c:   next = RETURN_15c;
        GOT_50c:   next = RETURN_20c;
        RETURN_10c:  next = RETURN_10c;
        RETURN_15c:  next = RETURN_15c;
        RETURN_20c:  next = RETURN_20c;
        RETURN_5c:   next = IDLE;
        RETURN_10c:  next = IDLE;
        default:    next = IDLE;
    endcase
end

Combinational output assignment

assign DC = (state == GOT_30c || state == GOT_35c ||
            state == GOT_40c || state == GOT_45c ||
            state == GOT_50c);
assign DN = (state == RETURN_5c);
assign DD = (state == RETURN_20c || state == RETURN_15c ||
            state == RETURN_10c);
endmodule
Simulation of Moore Vender

![Simulation of Moore Vender](image)

**State**
- idle
- got5c
- got20c
- got15c
- got45c
- rtn15
- rtn5
- idle

**Output**
- 10¢
- 5¢
FSM Output Glitching

- FSM state bits may not transition at precisely the same time
- Combinational logic for outputs may contain hazards
- Result: your FSM outputs may glitch!

during this state transition...

...the state registers may transition like this...

...causing the DC output to glitch like this!

assign DC = (state == GOT_30c || state == GOT_35c || state == GOT_40c || state == GOT_45c || state == GOT_50c);

If the soda dispenser is glitch-sensitive, your customers can get a 20-cent soda!
Registered FSM Outputs are Glitch-Free

- **Move output generation into the sequential always block**
- **Calculate outputs based on next state**
- **Delays outputs by one clock cycle. Problematic in some application.**

```verbatim
reg DC,DN,DD;

// Sequential always block for state assignment
always @ (posedge clk or negedge reset) begin
  if (!reset) state <= IDLE;
  else if (clk) state <= next;

  DC <= (next == GOT_30c || next == GOT_35c ||
          next == GOT_40c || next == GOT_45c || next == GOT_50c);
  DN <= (next == RETURN_5c);
  DD <= (next == RETURN_20c || next == RETURN_15c ||
          next == RETURN_10c);
end
```
Where should CLK come from?

• **Option 1: external crystal**
  – Stable, known frequency, typically 50% duty cycle

• **Option 2: internal signals**
  – Option 2A: output of combinational logic
    - No! If inputs to logic change, output may make several transitions before settling to final value \(\rightarrow\) several rising edges, not just one! Hard to design away output glitches…
  – Option 2B: output of a register
    - Okay, but timing of CLK2 won’t line up with CLK1