L6: FSMs and Synchronization

Lecture material courtesy of Rex Min

Lecture notes prepared by Professor Anantha Chandrakasan
Asynchronous Inputs in Sequential Systems

What about external signals?

Can’t guarantee setup and hold times will be met!

When an asynchronous signal causes a setup/hold violation...

Q: Which cases are problematic?
Asynchronous Inputs in Sequential Systems

All of them can be, if more than one happens simultaneously within the same circuit.

Idea: ensure that external signals directly feed exactly one flip-flop

This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?
Preventing metastability turns out to be an impossible problem.

High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly.

Solution to metastability: allow time for signals to stabilize.

How many registers are necessary?

- Depends on many design parameters (clock speed, device speeds, ...)
- In 6.111, one or maybe two synchronization registers is sufficient.
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.
Timing requirements for FSM are identical to any generic sequential system with feedback.

Minimum Clock Period

\[ T > T_{cq} + T_{\text{logic}} + T_{su} \]

Minimum Delay

\[ T_{cq,cd} + T_{\text{logic,cd}} > T_{\text{hold}} \]
Two Types of FSMs

Moore and Mealy FSMs are distinguished by their output generation.

**Moore FSM:**
- Inputs: $x_0 \ldots x_n$
- Outputs: $y_k = f_k(S)$

**Mealy FSM:**
- Inputs: $x_0 \ldots x_n$
- Outputs: $y_k = f_k(S, x_0 \ldots x_n)$

Direct combinational path!
A level-to-pulse converter produces a single-cycle pulse each time its input goes high.

In other words, 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.
State Transition Diagrams

- Block diagram of desired system:

```
unsynchronized user input
```

- State transition diagram is a useful FSM representation and design aid

```
00
Low input, Waiting for rise
P = 0

01
Edge Detected!
P = 1

11
High input, Waiting for fall
P = 0

“if L=1 at the clock edge, then jump to state 01.”

“if L=0 at the clock edge, then stay in state 00.”

This is the output that results from this state. (Moore or Mealy?)

Binary values of states

L=0

L=1

CLK

D Q

D Q

L

P

Level to Pulse FSM

Synchronizer

Edge Detector

Moore or Mealy?
Logic Derivation for a Moore FSM

- 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>$S_1$</td>
<td>$S_0$</td>
<td>$L$</td>
<td>$S_1^+$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Combinational logic may be derived by Karnaugh maps

\[
\begin{align*}
S_1^+ &= LS_0 \\
S_0^+ &= L \\
\end{align*}
\]

\[
P = \overline{S_1}S_0
\]
Moore Level-to-Pulse Converter

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

\[ y_k = f_k(S) \]

inputs \( x_0 \ldots x_n \)

outputs

\[ S_1^+ = LS_0 \]
\[ S_0^+ = L \]
\[ P = \bar{S}_1 S_0 \]
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. After the transition to $S=1$ and as long as $L$ remains at 1, this output is 0.
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

- Remember that the difference is in the output:
  - Moore outputs are based on state only
  - Mealy outputs are based on state and input
  - Therefore, 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
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...
State Reduction

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

There are two duplicates in our original diagram.

17 states 5 state bits 15 states 4 state bits
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)

**FSM Diagram:**
- Comb. Logic
- State Register
- Comb. Logic

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

**State register defined with sequential always block**

always @ (posedge clk or negedge reset)
    if (!reset)   state <= IDLE;
    else          state <= next;
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_5c;
        RETURN_15c: next = RETURN_5c;
        RETURN_20c: next = RETURN_10c;
        RETURN_5c: 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

State

idle got5c got15c got20c got45c rtn15 rtn5 idle

Output

C 10¢ 5¢
always @ (state or N or D or Q) begin

DC = 0; DD = 0; DN = 0; // defaults

GOT_5c: begin
    DC = 1; next = IDLE;
end

GOT_10c: begin
    DC = 1; next = RETURN_5c;
end

GOT_15c: begin
    DC = 1; next = RETURN_10c;
end

GOT_20c: begin
    DC = 1; next = RETURN_15c;
end

GOT_25c: begin
    DC = 1; next = RETURN_20c;
end

RETURN_20c: begin
    DD = 1; next = RETURN_10c;
end

RETURN_15c: begin
    DD = 1; next = RETURN_5c;
end

RETURN_10c: begin
    DD = 1; next = IDLE;
end

RETURN_5c: begin
    DN = 1; next = IDLE;
end

default: next = IDLE;
endcase
end
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!**

```c
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

```
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
```
A Mealy machine can eliminate states devoted solely to holding an output value.
module mealyVender (N, D, Q, DC, DN, DD, clk, reset, state);
  input N, D, Q, clk, reset;
  output DC, DN, DD;
  reg DC, DN, DD;

  output [3:0] state;
  reg [3:0] state, 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 RETURN_20c = 6;
  parameter RETURN_15c = 7;
  parameter RETURN_10c = 8;
  parameter RETURN_5c = 9;

  // Sequential always block for state assignment
  always @ (posedge clk or negedge reset)
    if (!reset) state <= IDLE;
    else state <= next;
always @ (state or N or D or Q) begin
    DC = 0; DN = 0; DD = 0; // defaults
    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) begin
                     DC = 1; next = IDLE;
                 end
            else if (D) next = GOT_15c;
            else if (N) next = GOT_10c;
            else next = GOT_5c;
        GOT_10c: if (Q) begin
                     DC = 1; next = RETURN_5c;
                 end
            else if (D) next = GOT_20c;
            else if (N) next = GOT_15c;
            else next = GOT_10c;
        GOT_15c: if (Q) begin
                     DC = 1; next = RETURN_10c;
                 end
            else if (D) begin
                     DC = 1; next = RETURN_5c;
                 end
            else if (N) begin
                     DC = 1; next = IDLE;
                 end
            else next = GOT_25c;
        GOT_20c: if (Q) begin
                     DC = 1; next = RETURN_15c;
                 end
            else if (D) begin
                     DC = 1; next = IDLE;
                 end
            else if (N) next = GOT_25c;
            else next = GOT_20c;
        RETURN_20c: begin
                        DD = 1; next = RETURN_10c;
                     end
        RETURN_15c: begin
                        DD = 1; next = RETURN_5c;
                     end
        RETURN_10c: begin
                        DD = 1; next = IDLE;
                     end
        RETURN_5c: begin
                        DN = 1; next = IDLE;
                     end
        default: next = IDLE;
    endcase
endmodule

For state GOT_5c, output DC is only asserted if Q=1
Simulation of Mealy Vendor

State

idle  got5c  got20c  rtn5  idle

Output

C  10¢  5¢

(note: outputs should be registered)
Delay Estimation: Simple RC Networks

(a) Low-to-high

(b) High-to-low

\[ V_{out} = V_D + V_D \left( 1 - e^{-t/\tau} \right) \]

\[ t_p = \ln(2) \tau = 0.69 RC \]
Clocks are Not Perfect: Clock Skew

Combinational Logic

In

D
Q

Combinational Logic

Q

D

CLout

ClkD

Wire delay

CLK

CLKD

δ>0

\[ T > T_{cq} + T_{logic} + T_{su} - \delta \]

\[ T_{cq,cd} + T_{logic,cd} > T_{hold} + \delta \]
Positive and Negative Skew

(a) Positive skew

Launching edge arrives before the receiving edge

(b) Negative skew

Receiving edge arrives before the launching edge

Clocks are Not Perfect: Clock Jitter

\[ T_{CLK} - 2t_{jitter} > t_{c-q} + t_{logic} + t_{su} \]

or

\[ T > t_{c-q} + t_{logic} + t_{su} + 2t_{jitter} \]
Summary

- Synchronize all asynchronous inputs
  - Use two back to back registers

- Two types of Finite State Machines introduced
  - **Moore** – outputs are a function of current state
  - **Mealy** – outputs a function of current state and input

- A standard template can be used for coding FSMs

- Register outputs of combinational logic for critical control signals

- Clock skew and jitter are important considerations