Sequential Logic Design

Controller Design

Five step controller design process

1. Capture the FSM
   - Create an FSM that describes the desired behavior of the controller.

2. Create the architecture
   - Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM inputs and outputs being the next state bits and the FSM outputs.

3. Encode the states
   - Assign a unique binary number to each state. Each binary number representing a state is known as an encoding. Any encoding will do as long as each state has a unique encoding.

4. Create the state table
   - Create a truth table for the combinational logic such that the logic will generate the correct FSM outputs and next state signals. Ordering the inputs with state bits first makes the truth table describe the state behavior, so the table is a state table.

5. Implement the combinational logic
   - Implement the combinational logic using any method.

Step 1: Capture the FSM
- Already done

Step 2: Create architecture
- 2-bit state register (for 4 states)
- Input b, output x
- Next state signals n1, n0

Step 3: Encode the states
- Any encoding with each state unique will work

Controller Design: Laser Timer Example

Inputs: b; Outputs: x

Combinational logic

State register

 clk
 s1 s0

n1 n0
Sequential Logic Design
Controller Design: Laser Timer Example (cont)

**Step 4: Create state table**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s1 s0 b</td>
</tr>
<tr>
<td>Off</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>On1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>On2</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>On3</td>
<td>1 1 0 0</td>
</tr>
</tbody>
</table>

**Step 5: Implement combinational logic**

\[ x = s1' + s0 \]
\[ n1 = s1's0b' + s1's0b + s1s0'b' + s1s0'b \]
\[ n0 = s1's0 + s1s0' \]

Understanding the Controller’s Behavior
Sequential Logic Design
Simplifying Notations

- FSMs
  - Assume unassigned output implicitly assigned 0
- Sequential circuits
  - Assume unconnected clock inputs connected to same external clock

Sequential Logic Design
Controller Example: Secure Car Key
(from earlier example)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s2 s1 s0 a</td>
<td>r n2 n1 n0</td>
</tr>
<tr>
<td>Wait</td>
<td>0 0 0 1 0 0 0 1</td>
</tr>
<tr>
<td>K1</td>
<td>0 1 0 1 0 0 1 1</td>
</tr>
<tr>
<td>K2</td>
<td>0 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>K3</td>
<td>0 1 1 0 1 0 0 0</td>
</tr>
<tr>
<td>K4</td>
<td>1 0 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

Sequential Logic Design
FSM Example: Code Detector

If we changed the state encoding for the secure car key design to the following, would this affect the final output?
1. Yes
2. No

Sequential Logic Design
In Class Exercise: Button Press Synchronizer

Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button actually pressed
- We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller
Sequential Logic Design

FSM Transitions

- Is the following FSM valid?
  1. Yes
  2. No

Sequential Logic Design

Common Pitfalls Regarding Transition Properties

- Only one condition should be true
  - For all transitions leaving a state
  - Else, which one?

- One condition must be true
  - For all transitions leaving a state
  - Else, where go?

- Can verify using Boolean algebra
  - Only one condition true: AND of each condition pair (for transitions leaving a state) should equal 0 → proves pair can never simultaneously be true
  - One condition true: OR of all conditions of transitions leaving a state) should equal 1 → proves at least one condition must be true

Sequential Logic Design

Evidence that Pitfall is Common

- Recall code detector FSM
  - We “fixed” a problem with the transition conditions
  - Do the transitions obey the two required transition properties?
    - Consider transitions of state Start, and the “only one true” property

Sequential Logic Design

Flip-Flop Set and Reset Inputs

- Some flip-flops have additional inputs
  - Synchronous reset: clears Q to 0 on next clock edge
  - Asynchronous reset: clear Q to 0 immediately (not dependent on clock edge)

- Example timing diagram shown
Sequential Logic Design
Initial State of a Controller

- All our FSMs had initial state.
- But our sequential circuit designs did not.
- Can accomplish using flip-flops with reset/set inputs.
  - Shown circuit initializes flip-flops to 01.
- Circuits typically have power on reset circuitry to automatically reset circuit on power up.

Sequential Logic Design
Non-Ideal Flip-Flop Behavior

- Can't change flip-flop input too close to clock edge.
  - Setup time: time that D must be stable before edge.
    - Else, stable value not present at internal latch.
  - Hold time: time that D must be held stable after edge.
    - Else, new value doesn't have time to loop around and stabilize in internal latch.

Sequential Logic Design
Metastability

- Violating setup/hold time can lead to bad situation known as metastable state.
- Metastable state: Any flip-flop state other than stable 1 or 0.
  - Eventually settles to one or other, but we don't know which.
- For internal circuits, we can make sure observe setup time.
- But what if input comes from external (asynchronous) source, e.g., button press?
- Partial solution.
  - Insert synchronizer flip-flop for asynchronous input.
    - Special flip-flop with very small setup/hold time.
  - Doesn't completely prevent metastability.
Other flip-flop types

- SR flip-flop: like SR latch, but edge triggered
- J K flip-flop: like SR (S→J, R→K)
  - But when J K=11, toggles
  - 1→0, 0→1
- T flip-flop: J K with inputs tied together
  - Toggles on every rising clock edge
- Previously utilized to minimize logic outside flip-flop
  - Today, minimizing logic to such extent is not as important
  - D flip-flops are thus by far the most common