Sequential logic implementation

- Sequential circuits
- Primitive sequential elements
- Combinational logic
- Models for representing sequential circuits
  - Finite state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)
- Basic sequential circuits
- Shift registers
- Counters
- Design procedure
  - State diagrams
  - State transition table
  - Next state functions

Abstraction of state elements

- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic

Forms of sequential logic

- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)

Finite state machine representations

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements
- Sequential logic: sequences through a series of states
  - Based on sequence of values on input signals
  - Clock period identifies element of sequence

Example finite state machine diagram

- Combination link from introduction to course

Can any sequential system be represented with a state diagram?

- Shift register
  - Input value shown on transition arc
  - Output values shown within state node
Counters are simple finite state machines

- Counters
  - process through well-defined sequence of states in response to enable
  - Many types of counters: binary, BCD, Gray-code
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000

How do we turn a state diagram into logic?

- Counter
  - 3 flip-flops to hold state
  - Logic to compute next state
  - Clock signal controls when flip-flop memory can change
  - Waiting enough for combinational logic to compute new value
  - Don't wait too long as that below performance

FSM design procedure

- Start with counters
  - Simple because output is just state
  - Simple because no choice of next state based on input

- State diagram to state transition table
  - Tabular form of state diagram
  - Like truth-table
  - State encoding
  - Decide on representation of states
  - For counters it is simpler: just its value

- Implementation
  - Flip-flop for each state bit
  - Combinational logic based on encoding

FSM design procedure: state diagram to encoded state transition table

- Table form of state diagram
  - Like a truth-table (specify output for all input combinations)
  - Encoding of states: easy for counters -- just use value

Implementation

- Flip-flop for each state bit
- Combinational logic based on encoding

Implementation (cont'd)

- Combinational logic: building block for sequential logic
  - D-flip-flop
    - Two-level logic capability like AND (i.e., 2 product terms)
### Another example

- **Shift register**
  - Input determines next state

#### State Transition Table

<table>
<thead>
<tr>
<th>Dn</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>B1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### More complex counter example

- **Complex counter**
  - Repeats 5 states in sequence
  - Not a binary number representation

#### State Transition Diagram

- **Step 1:** Derive the state transition diagram
  - Count sequence: 000, 010, 011, 101, 110

- **Step 2:** Derive the state transition table from the state transition diagram

#### Note

- Transitions that arise from the unused state codes

### More complex counter example (cont'd)

- **Step 3:** K-map for next state functions

#### K-Maps

- C = \(A \oplus B \oplus C\)
- B = \(A \oplus B \oplus C\)
- A = \(B \oplus C\)

### Self-starting counters (cont'd)

- **Re-derived state transition table from don't care assignment**

#### State Machine Model

- **Values stored in registers represent the state of the circuit**
- **Combinational logic computes:**
  - Next state
  - Outputs
    - Function of current state and inputs
    - Function of current state only (Mealy machine)
    - Function of current state only (Moore machine)

#### Inputs - Outputs (Example)

- **Implementations:**
  - Sequential Circuit
  - Mealy Machine
  - Moore Machine
State machine model (cont’d)

- States: S1, S2, ..., S5
- Inputs: l1, l2, ..., l5
- Outputs: O1, O2, ..., O5
- Transition function: f(S, I)
- Output function: N(E) or O(E

Example: ant brain (Ward, MIT)

- Sensors: L and R antennae, L if I’m touching wall
- Actuators: F - forward step, T/UR - turn left/right slightly
- Goal: find way out of maze
- Strategy: keep the wall on the right

Ant behavior

- In following wall, reaching Goal, turn right
- In following wall, not reaching Goal, turn left
- Touching wall, turn left
- Wall in front, turn left
- LOST: return to 1st state

Designing an ant brain

- State diagram

Synthesizing the ant brain circuit

- Include states using a set of state variables
- Bolean circuit - very small, costly, speed
- Use transition truth table
  - Define next state function for each state variable
  - Define output function for each output
- Implement next state and output functions using combinational logic
- 2-level logic (NAND/AND)
- multi-level logic
- Next state and output functions can be optimized together

Transition truth table

- Using symbolic states and outputs

- States: L, R
**Synthesis**

- 5 states: at least 3 state variables required (x, y, z)
- State assignment: in this case, arbitrarily chosen

<table>
<thead>
<tr>
<th>state</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>0</td>
<td>0</td>
<td>x, y, z</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>s1</td>
<td>0</td>
<td>1</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>s2</td>
<td>1</td>
<td>0</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>s3</td>
<td>0</td>
<td>1</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>s4</td>
<td>1</td>
<td>1</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
</tbody>
</table>

**Synthesis of next state and output functions**

<table>
<thead>
<tr>
<th>state</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>0</td>
<td>0</td>
<td>x, y, z</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>s1</td>
<td>0</td>
<td>1</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>s2</td>
<td>1</td>
<td>0</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>s3</td>
<td>0</td>
<td>1</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>s4</td>
<td>1</td>
<td>1</td>
<td>0, 1, 0</td>
<td>1, 0</td>
</tr>
</tbody>
</table>

**Circuit implementation**

- Outputs are a function of the current state only: Moore machine

**Don’t cares in FSM synthesis**

- What happens to the “don’t care” states (011, 110, 111)?
- They were exploited as don’t cares to minimize the logic

**State minimization**

- Fewer states may mean fewer state variables
- Alphabet reduction may generate many redundant states
- Two states are equivalent if they are impossible to distinguish from the outputs of the FSM, i.e., for any input sequence the outputs are the same
- Two conditions for two states to be equivalent:
  1. output must be the same in both states
  2. must transition to equivalent states for all input combinations

**Ast brain revisited**

- Any equivalent states?
New improved brain

- Merge equivalent B and C states
- Behavior is exactly the same as the 5-state brain
- We now need only 2 state variables rather than 3

New brain implementation

Mealy vs. Moore machines

- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our unit brains is a Moore machine
  - output does not react immediately to input change
  - outputs have immediate reaction to inputs
- as inputs change, so does next state, doesn’t commit until closing event

Specifying outputs for a Mealy machine

- Output is function of state and inputs
  - specify output on transition accross between states
- examples: sequence detector for 01 or 10

Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - different outputs on each (n^2) rather than states (n)
- Moore machines are easier to use
  - outputs change at edge (always one code later)
  - in Mealy machines, input change can cause output change as soon as logic is done — no problem when two machines are interconnected — asynchronous feedback
- Mealy machines read faster to inputs
  - react in same cycle — don’t need to wait for clock
  - in Moore machines, more logic may be necessary to decode state into outputs — more gate delays after
Mealy and Moore examples

- Recognize A,B = 0,1
  - Mealy or Moore?

Registered Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
- registered state AND outputs
- avoid “glitchy” outputs
- easy to implement in FUs
- Moore machine with no output-decoding
- outputs computed on transition to next state rather than after entering
- view outputs as expanded state vector

Example: vending machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change

Example: vending machine (cont'd)

- Suitable abstract representation
  - tabulate typical input sequences:
    - Tickets
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs N, D, reset
    - output open slave
  - assumptions:
    - assume N and D asserted for one code
    - each state has a self loop
    - for N = D = 0 (no coins)
  - Minimize number of states - same states whenever possible

Example: vending machine (cont'd)

- Minimize number of states - same states whenever possible
- Transition table
Example: vending machine (cont’d)

**Unlikely encode states**

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>new state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0</td>
<td>Q1</td>
<td>D</td>
<td>Qn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Mapping to logic**

- \( D1 = \overline{Q1} + D + \overline{Q0} \)
- \( D2 = Q1 + \overline{D} + Q0 \)
- \( C1 = Q0 \)
- \( \text{open} = Q1 \)

Example: vending machine (cont’d)

**One-hot encoding**

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>new state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0</td>
<td>Q1</td>
<td>D</td>
<td>Qn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Equivalent Mealy and Moore state diagrams

- **Mealy machine**
  - outputs associated with state
  - \( D1 = \overline{Q1} + D + \overline{Q0} \)
  - \( D2 = Q1 + \overline{D} + Q0 \)
  - \( C1 = Q0 \)
  - \( \text{open} = Q1 \)

- **Moore machine**
  - outputs associated with transitions

Example: traffic light controller

- A four-way intersection is controlled by a little used form real
- Drivers can see the presence of cars waiting on the form real
- With no cars on form real, lights remain green in highway direction
- If vehicle on form real, highway lights go from Green to Yellow to Red, allowing the form real lights to become green
- These stay green only as long as a form real car is detected but never longer than a set interval
- When there are no cars, form real lights transition from Green to Yellow to Red, allowing highway to return to green
- Even if form real vehicles are waiting, highway gate at least a set interval as green
- Assume no cars have an interval that generates:
  - a signal (T1) after a short time has elapsed and, a signal (T1) after a long time has elapsed
  - T1 to be used for turning amber lights and T1 for green lights
  - else time is started to expiration for a valid signal; the flashing red lights and T1

Footnote: Interrupts

- C, T5, and T1 are asynchronous signals
- Such signals coming from independent devices are often called interrupts
- Interrupts are a memory mechanism for S.O. devices to communicate status to a CPU
  - "Data transfer complete"
  - "Timer run-out"
  - "Device needs attention"
  - "Error has occurred"
  - "Power button pressed"
Example: traffic light controller (cont')

- Highway/interroad intersection

Example: traffic light controller (cont')

- Tabulation of inputs and outputs

<table>
<thead>
<tr>
<th>Inputs/Outputs</th>
<th>Inputs/Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highway green</td>
<td>Traffic light</td>
</tr>
<tr>
<td>Traffic light</td>
<td>Highway green</td>
</tr>
</tbody>
</table>

Example: traffic light controller (cont')

- Tabulation of unique states - some light configurations imply others

- State diagram

Example: traffic light controller (cont')

- Generate state table with symbolic states

- Logic for different state assignments

- Vending machine example (PLD mapping)
Vending machine (cont'd)

- OFF = QiQi creates a combinational delay after Qi and Q' change
- This can be corrected by delaying, i.e., more flipflops and logic through each other to improve delay
- OFF = reset(Qi + Q) = Q = Q'N + Q'R + Q'R + Q'R = reset
- Implementation requires a synchronous Mealy machine
- It is common for programmable arrays to have 1 at end of logic

Finite state machine optimization

- State minimization
  - fewer states require fewer state bits
  - fewer bits require fewer logic equations
  - each state's inputs, outputs
  - state encoding with fewer bits has fewer equations to implement
  - however, each may be more complex
  - state encoding with more bits (e.g., one-hot) has simpler equations
  - complexity greatly related to complexity of state diagram
  - input/output encoding may or may not be under designer control

Algorithmic approach to state minimization

- Goal: identify and combine states that have equivalent behavior
- Equivalent states:
  - same output
  - for all input combinations, states transition to same or equivalent states
- Algorithmic steps:
  1. place all states in one set
  2. initially partition set based on output behavior
  3. successively partition resulting subsets based on next state transitions
  4. repeat (3) until no further partitioning is required
- states left in the same set are equivalent
- polynomial time procedure

State minimization example

- Sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>State Sequence</th>
<th>Present State</th>
<th>Next State 1</th>
<th>Next State 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>012</td>
<td>03</td>
<td>02</td>
<td>10</td>
</tr>
<tr>
<td>110</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>10</td>
</tr>
<tr>
<td>001</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>011</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>101</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>10</td>
</tr>
<tr>
<td>111</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>10</td>
</tr>
<tr>
<td>000</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>00</td>
</tr>
<tr>
<td>010</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>012</td>
<td>02</td>
<td>03</td>
<td>10</td>
</tr>
</tbody>
</table>

Method of successive partitions

<table>
<thead>
<tr>
<th>State Sequence</th>
<th>Present State</th>
<th>Next State 1</th>
<th>Next State 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final</td>
<td>55</td>
<td>51</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>51</td>
<td>53</td>
<td>54</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>55</td>
<td>56</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>50</td>
<td>53</td>
<td>54</td>
<td>0</td>
</tr>
</tbody>
</table>

(56) is equivalent to 55
(55) is equivalent to 56
(54) is equivalent to 56
(50) is equivalent to 55
Minimized FSM

State minimized sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>Input</th>
<th>Present State</th>
<th>Next State</th>
<th>X</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Minimized FSM

- Implication-division method
  1. Cross out incompatible states based on outputs
  2. Then cross out more cells if implied states are already crossed out

Minimizing incompletely specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But it's not transitive when don't-care states are present
  
  e.g., state output
  
<table>
<thead>
<tr>
<th>state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
</tr>
</tbody>
</table>

- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Minimizing states may not yield best circuit

Example: edge detector - outputs 1 when last two inputs changes from 0 to 1

<table>
<thead>
<tr>
<th>Q</th>
<th>Q'</th>
<th>Q''</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q'' = X (Q or Q')
Q' = X Q
Q'' = X Q'

Another implementation of edge detector

"Addic" solution • not minimal but cheap and fast
State assignment

- Choose bit vectors to assign to each "hemibit" state
  - with n state bits for m states there are \(2^n\) / \(2^m\) possibilities
  - 2^m cases possible for bit set, 2^m = 2m
  - hope number even for small values of n and m
  - interdicted for state machines of any size
  - heuristics are necessary for practical solutions
  - optimize some metric for the combined cost of logic size and number of flip-flops
  - speed (depth of logic and fanout)
  - dependencies (decomposition)

State assignment strategies

- Realistic strategies
  - sequential - just number states as they appear in the state table
  - random - pick random cases
  - one-hot - use as many state bits as there are states
  - output - use outputs to help encode states
  - heuristic - rules of thumb that seem to work in most cases
  - No guarantee of optimality - another intractable problem

One-hot state assignment

- Simple
  - easy to encode
  - easy to lookup
- Small logic functions
  - each state function requires only predecessor state bits as input
- Good for programmable devices
  - lots of flip-flops readily available
- Simple functions with small support (signals dependent upon)
- Inapplicable for large machines
  - too many states require too many flip-flops
  - decompose FSM into smaller pieces that can be one-hot encoded
- Many slight variations to one-hot
  - one-hot + static

Heuristics for state assignment

- Adjacent states to states that share a common next state
  - group 5's in next state map
  - \( q = a \times b + c \)
- Adjacent states to states that share a common ancestor state
  - group 5's in next state map
  - \( q = b \times c \)
- Adjacent states to states that have a common output behavior
  - group 5's in output map
  - \( q = d \times c \)

General approach to heuristic state assignment

- All current methods are variants of this
  - 1) determine which states "hit" each other (weighted pairs)
  - 2) generate constraints on codes (which should be in same cube)
  - 3) piece codes on Boolean cube so as to minimize constraints satisfied (weighted sum)
  - Different weights make sense depending on whether we are optimizing for two-level or multi-level forms
  - Can't consider all possible embeddings of state clusters in Boolean cube
  - Heuristics for ordering embedding
  - To prune search for best embedding
  - Expand cube (more state bits) to satisfy more constraints

Output-based encoding

- Reuse outputs as state bits - use outputs to help distinguish states
- Use create new functions for state logic when output can serve as well
- fits in nicely with synchronous multiplex implementations

<table>
<thead>
<tr>
<th>States</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

Output patterns are subject to states, we do not include all of them above
(see below for the output patterns)
Current state assignment approaches

- For light encodings using close to the minimum number of state bits
- Best of 10 random seems to be adequate (averages as well as heuristics)
- Heuristic approaches are not even close to optimality
- Used in custom chip design

One-hot encoding

- Easy for small state machines
- Generates small equations with easy-to-estimate complexity
- Common in FPGAs and other programmable logic

Output-based encoding

- Hard-wired, no tools
- Most common approach taken by human designers
- Yields very small circuits for most FSMs

Sequential logic implementation summary

- Models for representing sequential circuits
- Abstraction of sequential elements
- Finite state machines and their state diagrams
- Inputs/outputs
- Mealy, Moore, and synchronous Mealy machines

Finite state machine design procedure

- Deriving state diagram
- Deriving state transition table
- Determining next state and output functions
- Implementing combinational logic

Implementation of sequential logic

- State minimization
- State assignment
- Support in programmable logic devices