Sequential logic implementation

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines (Mealy and More)
  - 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
Abstraction of state elements (Cont.)

- Special Case: No inputs
- Example: Traffic light with no pedestrian control button
Abstraction of state elements (Cont.)

- No explicit outputs
  - Output values shown within state node
  - Example: Counters with LD, EN and CLR inputs
Abstraction of state elements (Cont.)

• No explicit input and outputs
  - Output values shown within state node
  - Example: Counters with no LD, EN and CLR inputs
Forms of sequential logic

• Asynchronous sequential logic - state changes occur whenever state inputs change (elements may be simple wires or delay elements)
  (Hard to design due to race condition, Not used in main domain digital design). Do not deal with it in this class. Rarely used by the industry.

• 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 defines elements of sequence

No explicit output value
Output values shown within state node
Example finite state machine diagram

- Combination lock from the introduction

```
reset

S1
  closed, mux=C1
  equal & new
  not new

S2
  closed
  equal, mux=C2
  not new

S3
  closed
  equal, mux=C3
  not new

OPEN

DATA-PATH
```

Controller
Can any sequential system be represented with a state diagram?

- **Shift register**
  - input value shown on transition arcs
  - output values shown within state node

---

**FSM with input and no explicit outputs**
Counters are simple finite state machines

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

```
000 ----> 001 ----> 010 ----> 011
       ^                ^
       |                |
       ↓                ↓
111 ----> 110 ----> 101 ----> 100
```

3-bit up-counter
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
    - wait long enough for combinational logic to compute new value
    - don't wait too long as that is low 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 a truth-table
- State encoding (prepare inputs)
  - decide on representation of states
  - for counters it is simple: 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**

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

<table>
<thead>
<tr>
<th>current state</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001 1</td>
</tr>
<tr>
<td>010</td>
<td>010 2</td>
</tr>
<tr>
<td>011</td>
<td>011 3</td>
</tr>
<tr>
<td>100</td>
<td>100 4</td>
</tr>
<tr>
<td>101</td>
<td>101 5</td>
</tr>
<tr>
<td>110</td>
<td>110 6</td>
</tr>
<tr>
<td>111</td>
<td>111 7</td>
</tr>
<tr>
<td>111</td>
<td>000 0</td>
</tr>
</tbody>
</table>

**Combinational Logic**

**Outputs**

**Storage Elements**
Implementation

- D flip-flop for each state bit
- Combinational logic based on encoding

\[
\begin{array}{c|ccc|ccc}
C3 & C2 & C1 & N3 & N2 & N1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{align*}
N1 & := C1' \\
N2 & := C1C2' + C1'C2 \\
& := C1 \ xor \ C2 \\
N3 & := C1C2C3' + C1'C3 + C2'C3 \\
& := (C1C2) \ xor \ C3
\end{align*}
\]
Implementation

- D flip-flop for each state bit
- Combinational logic based on encoding

<table>
<thead>
<tr>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>N3</th>
<th>N2</th>
<th>N1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 0 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N1 := C1'
N2 := C1C2' + C1'C2
    := C1 xor C2
N3 := C1C2C3' + C1'C3 + C2'C3
    := C1C2C3' + (C1' + C2')C3
    := (C1C2) xor C3

Storage Elements

 Outputs

CLK

"1"

Combinational
Logic

D Q

D Q

D Q

OUT1

OUT2

OUT3

D Q

N1

C1

N2

C2

N3

C3

Implementation

• D flip-flop for each state bit
• Combinational logic based on encoding
Another example

• Shift register
  - input determines next state

<table>
<thead>
<tr>
<th>In</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>N1</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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

- **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**

  ![State Transition Diagram]

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C  B  A</td>
<td>C+  B+  A+</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0  1  0</td>
</tr>
<tr>
<td>0  0  1</td>
<td>-  -  -</td>
</tr>
<tr>
<td>0  1  0</td>
<td>0  1  1</td>
</tr>
<tr>
<td>0  1  1</td>
<td>1  0  1</td>
</tr>
<tr>
<td>1  0  0</td>
<td>-  -  -</td>
</tr>
<tr>
<td>1  0  1</td>
<td>1  1  0</td>
</tr>
<tr>
<td>1  1  0</td>
<td>0  0  0</td>
</tr>
<tr>
<td>1  1  1</td>
<td>-  -  -</td>
</tr>
</tbody>
</table>

  note the don't care conditions that arise from the unused state codes
More complex counter example (cont’d)

• Step 3: K-maps for next state functions

C+ := A
B+ := B' + A'C'
A+ := BC'
Self-starting counters

- **Start-up states**
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state
- **Self-starting solution**
  - design counter so that invalid states eventually transition to a valid state
  - may limit exploitation of don't cares
Self-starting counters (cont’d)

- Re-deriving state transition table from don't care assignment

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

C+ := A

B+ := B' + A'C'

A+ := BC'