Finite State Machine

1. Review on counter design
2. State Diagrams for FSM
3. Moore & Mealy Models
4. State Minimization
5. Examples
6. HDL for FSM
Sequential Logic Implementation

• Models for representing sequential circuits
  – Finite-state machines (Moore and Mealy)
  – Representation of memory (states)
  – Changes in state (transitions)

• Design procedure
  – State diagrams
  – State transition table
  – Next state functions
2 – STATE DIAGRAM FOR FSM
Finite State Machines

• Sequential circuits
  – primitive sequential elements
  – combinational logic
• Models for representing sequential circuits
  – finite-state machines (Moore and Mealy)
• Basic sequential circuits revisited
  – shift registers
  – counters
• Design procedure
  – state diagrams
  – state transition table
  – next state functions
• Hardware description languages
Abstraction of state elements

1. Divide circuit
   - combinational logic and state

2. Feedback loops
   - Localize
   - Break cycles

3. storage elements
   - sequential logic
Forms of sequential logic

1. Asynchronous sequential logic
   - state changes occur whenever state inputs change
   - elements may be simple wires or delay elements

2. Synchronous sequential logic
   - state changes occur in lock step across all storage elements
   - using a periodic waveform - the clock
Finite State Machine Representations

1. States
   - determined by possible values in sequential storage elements
2. Transitions
   - change of state
3. Clock
   - controls when state can change by controlling storage elements
4. Sequential logic
   - sequences through a series of states
   - based on sequence of values on input signals
   - clock period defines elements of sequence
Can any sequential system be represented with a state diagram?

- Shift register
  - input value shown on transition arcs
  - output values shown within state node
1- DESIGN A COUNTER
Counters are simple finite state machines

- Counters
  - proceed 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, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...
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
  – 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
Finite State Machine

FSM design procedure: state diagram to encoded state transition table – counter case 1

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

![State Diagram](image)

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

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

Verilog notation to show function represents an input to D-FF

<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</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</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>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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<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>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

N1 \(\leq C1'\)
N2 \(\leq C1C2' + C1'C2\)
\(\leq C1 \text{ xor} C2\)
N3 \(\leq C1C2C3' + C1'C3 + C2'C3\)
\(\leq (C1C2)C3' + (C1' + C2')C3\)
\(\leq (C1C2)C3' + (C1C2)'C3\)
\(\leq (C1C2) \text{ xor} C3\)
Back to the shift register - counter case 2

- 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>1</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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<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>
<td>1</td>
<td>0</td>
<td>1</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>1</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 - counter case 3

• 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

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

![K-maps](image)

\[ C^+ = A \]
\[ B^+ = B' + A'C' \]
\[ A^+ = BC' \]
Self-starting counters (cont’d) – counter case 3a

- 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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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
Activity

- 2-bit up-down counter (2 inputs)
  - direction: $D = 0$ for up, $D = 1$ for down
  - count: $C = 0$ for hold, $C = 1$ for count
Activity (cont’d)
Counter/shift-register model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - values of flip-flops
General state machine model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)
State machine model (cont’d)

- States: $S_1, S_2, ..., S_k$
- Inputs: $I_1, I_2, ..., I_m$
- Outputs: $O_1, O_2, ..., O_n$
- Transition function: $F_s(S_i, I_j)$
- Output function: $F_o(S_i)$ or $F_o(S_i, I_j)$
3 - MOORE & MEALY MACHINES
Mealy vs Moore Machines

- Mealy machines tend to have less states
  - different outputs on arcs \((n^2)\) rather than states \((n)\)
- Moore machines are safer to use
  - outputs change at clock edge (always one cycle later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback may occur if one isn’t careful
- Mealy machines react 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 clock edge
Comparison of Mealy and Moore machines (cont’d)

- Moore

- Mealy

- Synchronous Mealy
Outputs for a Moore machine

- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10

```
reset  | input | current state | next state | output
--- | --- | --- | --- | ---
1 | - | - | A | A
0 | 0 | A | B | 0
0 | 1 | A | C | 0
0 | 0 | B | D | 0
0 | 1 | B | E | 0
0 | 0 | C | E | 0
0 | 1 | C | E | 0
0 | 0 | D | E | 1
0 | 1 | D | C | 1
0 | 0 | E | B | 1
0 | 1 | E | D | 1
```
Outputs for a Mealy machine

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

<table>
<thead>
<tr>
<th>reset</th>
<th>input</th>
<th>current state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>
Registered Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
  - registered state AND outputs
  - avoids ‘glitchy’ outputs
  - easy to implement in PLDs

- Moore machine with no output decoding
  - outputs computed on transition to next state rather than after entering
  - view outputs as expanded state vector
Activity

A 3-bit sequence detector will detect 111 and 001. All the input and output will be reset after each sequence had been detected. An example of input/output sequence that satisfies the conditions of the network specifications is as follows;

\[
x = 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1
\]
\[
z = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0
\]

Draw the state transition diagram of the system using Mealy machine method.
Example: Ant Brain (Ward, MIT)

- **Sensors:**
  - L and R antennae, 1 if in touching wall

- **Actuators:**
  - F - forward step, TL/TR - turn left/right slightly

- **Goal:**
  - find way out of maze

- **Strategy:**
  - keep the wall on the right
Ant Behavior

A: Following wall, touching
Go forward, turning left slightly

B: Following wall, not touching
Go forward, turning right slightly

C: Break in wall
Go forward, turning right slightly

D: Hit wall again
Back to state A

E: Wall in front
Turn left until...

F: ...we are here, same as state B

G: Turn left until...

LOST: Forward until we touch something
Designing an Ant Brain

• State Diagram
Synthesizing the Ant Brain Circuit

• Encode States Using a Set of State Variables
  – Arbitrary choice - may affect cost, 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 (ROM/PLA/PAL)
  – Multi-level logic
  – Next state and output functions can be optimized together
Transition Truth Table

- Using symbolic states and outputs

<table>
<thead>
<tr>
<th>state</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOST</td>
<td>0</td>
<td>0</td>
<td>LOST</td>
<td>F</td>
</tr>
<tr>
<td>LOST</td>
<td>–</td>
<td>1</td>
<td>E/G</td>
<td>F</td>
</tr>
<tr>
<td>LOST</td>
<td>1</td>
<td>–</td>
<td>E/G</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>TL, F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>TL, F</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>–</td>
<td>E/G</td>
<td>TL, F</td>
</tr>
<tr>
<td>B</td>
<td>–</td>
<td>0</td>
<td>C</td>
<td>TR, F</td>
</tr>
<tr>
<td>B</td>
<td>–</td>
<td>1</td>
<td>A</td>
<td>TR, F</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Synthesis

- 5 states: at least 3 state variables required \((X, Y, Z)\)
  - State assignment (in this case, arbitrarily chosen)

<table>
<thead>
<tr>
<th>state (X, Y, Z)</th>
<th>next state (X', Y', Z')</th>
<th>outputs (F\ TR\ TL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>0 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>0 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0 1 1</td>
<td>0 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 1 0 0</td>
<td>1 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>0 1 1 0 1</td>
<td>0 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- it now remains to synthesize these 6 functions

- LOST - 000
- E/G  - 001
- A    - 010
- B    - 011
- C    - 100
## Synthesis of Next State and Output Functions

<table>
<thead>
<tr>
<th>state inputs</th>
<th>next state outputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X, Y, Z, L, R$</td>
<td>$X^+, Y^+, Z^+$</td>
<td>$F, TR, TL$</td>
</tr>
<tr>
<td>0 0 0 0 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 0 - 1</td>
<td>0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 0 1 -</td>
<td>0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>0 1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1 - 1</td>
<td>0 1 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1 1 -</td>
<td>0 1 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>0 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0 1 -</td>
<td>0 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 1 - 0</td>
<td>1 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>0 1 1 - 1</td>
<td>0 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 0 - 0</td>
<td>1 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 0 - 1</td>
<td>0 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

*e.g.*

\[
TR = X + Y Z
\]
\[
X^+ = X R' + Y Z R' = R' TR
\]
Circuit Implementation

- Outputs are a function of the current state only - Moore machine
Don't Cares in FSM Synthesis

- What happens to the "unused" states (101, 110, 111)?
- Exploited as don't cares to minimize the logic
  - If states can't happen, then don't care what the functions do
  - If states do happen, we may be in trouble

Ant is in deep trouble if it gets in this state.
State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis 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
Ant 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

<table>
<thead>
<tr>
<th>state (X,Y) inputs</th>
<th>next state outputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X', Y'</td>
<td>F</td>
<td>TR</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 -1</td>
<td>0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 1 -</td>
<td>0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 - 1</td>
<td>0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 1 -</td>
<td>0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1 -</td>
<td>0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 - 0</td>
<td>1 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 - 1</td>
<td>1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

![New Brain Implementation Diagram](image-url)
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:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - Draw state diagram:
    - Inputs: N, D, reset
    - Output: open chute
  - Assumptions:
    - Assume N and D asserted for one cycle
    - Each state has a self loop for N = D = 0 (no coin)
Example: Vending Machine (cont’d)

- Minimize number of states - reuse states whenever possible

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>next state</th>
<th>output open</th>
</tr>
</thead>
<tbody>
<tr>
<td>0¢</td>
<td>D 0</td>
<td>0¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>5¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>5¢</td>
<td>0 0</td>
<td>5¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>10¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>10¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>10¢</td>
<td>0 0</td>
<td>10¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>15¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>15¢</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>15¢</td>
<td>–</td>
<td>15¢</td>
<td>1</td>
</tr>
</tbody>
</table>

symbolic state table
### Example: Vending Machine (cont’d)

- Uniquely Encode States

<table>
<thead>
<tr>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 Q0 D N D1 D0</td>
<td>open</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 1 0 0 0 0</td>
<td>0 1 0 0 0 0</td>
</tr>
<tr>
<td>0 1 1 0 0 0</td>
<td>0 1 1 0 0 0</td>
<td>1 1 1 1 1 1</td>
<td>1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
Example: Vending Machine (cont’d)

- Mapping to Logic

\[
\begin{align*}
D1 &= Q1 + D + Q0 \cdot N \\
D0 &= Q0' \cdot N + Q0 \cdot N' + Q1 \cdot N + Q1 \cdot D \\
OPEN &= Q1 \cdot Q0
\end{align*}
\]
Example: Vending Machine (cont’d)

- One-hot Encoding

<table>
<thead>
<tr>
<th>Present state (Q3 Q2 Q1 Q0)</th>
<th>Inputs (D N)</th>
<th>Next state output (D3 D2 D1 D0)</th>
<th>Output (open)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>0 0</td>
<td>0 0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>- - - -</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0</td>
<td>0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>- - - -</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0</td>
<td>0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>- - - -</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>- -</td>
<td>1 0 0 0 1</td>
<td>1</td>
</tr>
</tbody>
</table>

D0 = Q0 D’ N’
D1 = Q0 N + Q1 D’ N’
D2 = Q0 D + Q1 N + Q2 D’ N’
D3 = Q1 D + Q2 D + Q2 N + Q3
OPEN = Q3
Equivalent Mealy and Moore State Diagrams

- Moore machine
  - outputs associated with state

Mealy machine
  - outputs associated with transitions
Example: Traffic Light Controller

- A busy highway is intersected by a little used farmroad
- Detectors C sense the presence of cars waiting on the farmroad
  - with no car on farmroad, light remain green in highway direction
  - if vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green
  - these stay green only as long as a farmroad car is detected but never longer than a set interval
  - when these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green
  - even if farmroad vehicles are waiting, highway gets at least a set interval as green
Example: Traffic Light Controller

- Assume you have an interval timer that generates:
  - a short time pulse (TS) and
  - a long time pulse (TL),
  - in response to a set (ST) signal.
  - TS is to be used for timing yellow lights and TL for green lights
Example: Traffic Light Controller (cont’)

- Highway/farm road intersection
Example: Traffic Light Controller (cont’)

• Tabulation of Inputs and Outputs

<table>
<thead>
<tr>
<th>inputs</th>
<th>description</th>
<th>outputs</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>place FSM in initial state</td>
<td>HG, HY, HR</td>
<td>assert green/yellow/red highway lights</td>
</tr>
<tr>
<td>C</td>
<td>detect vehicle on the farm road</td>
<td>FG, FY, FR</td>
<td>assert green/yellow/red highway lights</td>
</tr>
<tr>
<td>TS</td>
<td>short time interval expired</td>
<td>ST</td>
<td>start timing a short or long interval</td>
</tr>
<tr>
<td>TL</td>
<td>long time interval expired</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Tabulation of unique states – some light configurations imply others

<table>
<thead>
<tr>
<th>state</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>highway green (farm road red)</td>
</tr>
<tr>
<td>S1</td>
<td>highway yellow (farm road red)</td>
</tr>
<tr>
<td>S2</td>
<td>farm road green (highway red)</td>
</tr>
<tr>
<td>S3</td>
<td>farm road yellow (highway red)</td>
</tr>
</tbody>
</table>
Example: Traffic Light Controller (cont’)

- State Diagram

S0: HG
S1: HY
S2: FG
S3: FY
Example: Traffic Light Controller (cont’)

- Generate state table with symbolic states
- Consider state assignments

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Present State</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>C</td>
<td>TL</td>
<td>TS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>HG</td>
</tr>
<tr>
<td>–</td>
<td>0</td>
<td>–</td>
<td>HG</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>–</td>
<td>HG</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>0</td>
<td>HY</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>1</td>
<td>HY</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>–</td>
<td>FG</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>FG</td>
</tr>
<tr>
<td>–</td>
<td>1</td>
<td>–</td>
<td>FG</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>0</td>
<td>FY</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>1</td>
<td>FY</td>
</tr>
</tbody>
</table>

SA1: HG = 00, HY = 01, FG = 11, FY = 10
SA2: HG = 00, HY = 10, FG = 01, FY = 11
SA3: HG = 0001, HY = 0010, FG = 0100, FY = 1000 (one-hot)

Output encoding – similar problem to state assignment (Green = 00, Yellow = 01, Red = 10)
Logic for Different State Assignments

• SA1

\[ \text{NS1} = C \cdot TL' \cdot PS1 \cdot PS0 + TS \cdot PS1' \cdot PS0 + TS \cdot PS1 \cdot PS0' + C' \cdot PS1 \cdot PS0 + TL \cdot PS1 \cdot PS0 \]
\[ \text{NS0} = C \cdot TL \cdot PS1' \cdot PS0' + C \cdot TL' \cdot PS1 \cdot PS0 + PS1' \cdot PS0 \]

\[ \text{ST} = C \cdot TL \cdot PS1' \cdot PS0' + TS \cdot PS1' \cdot PS0 + TS \cdot PS1 \cdot PS0' + C' \cdot PS1 \cdot PS0 + TL \cdot PS1 \cdot PS0 \]
\[ H1 = PS1 \]
\[ F1 = PS1' \]

\[ H0 = PS1' \cdot PS0 \]
\[ F0 = PS1 \cdot PS0' \]

• SA2

\[ \text{NS1} = C \cdot TL \cdot PS1' + TS' \cdot PS1 + C' \cdot PS1' \cdot PS0 \]
\[ \text{NS0} = TS \cdot PS1 \cdot PS0' + PS1' \cdot PS0 + TS' \cdot PS1 \cdot PS0 \]

\[ \text{ST} = C \cdot TL \cdot PS1' + C' \cdot PS1' \cdot PS0 + TS \cdot PS1 \]
\[ H1 = PS0 \]
\[ F1 = PS0' \]

\[ H0 = PS1' \cdot PS0' \]
\[ F0 = PS1 \cdot PS0 \]

• SA3

\[ \text{NS3} = C' \cdot PS2 + TL \cdot PS2 + TS' \cdot PS3 \]
\[ \text{NS2} = TS \cdot PS1 + C \cdot TL' \cdot PS2 \]
\[ \text{NS1} = C \cdot TL \cdot PS0 + TS' \cdot PS1 \]
\[ \text{NS0} = C' \cdot PS0 + TL' \cdot PS0 + TS \cdot PS3 \]

\[ \text{ST} = C \cdot TL \cdot PS0 + TS \cdot PS1 + C' \cdot PS2 + TL \cdot PS2 + TS \cdot PS3 \]
\[ H1 = PS3 + PS2 \]
\[ F1 = PS1 + PS0 \]

\[ H0 = PS1 \]
\[ F0 = PS3 \]
Vending Machine Example
(PLD mapping)

\[
\begin{align*}
D0 &= \text{reset}'(Q0^'N + Q0N' + Q1N + Q1D) \\
D1 &= \text{reset}'(Q1 + D + Q0N) \\
\text{OPEN} &= Q1Q0
\end{align*}
\]
Vending Machine (cont’d)

- OPEN = Q1Q0 creates a combinational delay after Q1 and Q0 change
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay
- OPEN = reset'(Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)
  = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)
- Implementation now looks like a synchronous Mealy machine
  – Common for programmable devices to have FF at end of logic
Vending Machine (Retimed PLD Mapping)

$$\text{OPEN} = \text{reset}'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)$$
STATE MINIMIZATION
Finite State Machine Optimization

- Should look over this one
- Make sure you know the motivation for this
  - State Minimization
    - Try to merge state with same behavior so total # of states are less
  - State Encoding
    - Less combinational logic involved in implementing if correct state encoding is chosen.
    - Less Flip flops used if a more dense encoding is used.
      - # of flip flop could range from log2(n) to n
- Make sure you know the algorithm for both
Finite State Machine Optimization

• State Minimization
  – Fewer states require fewer state bits
  – Fewer bits require fewer logic equations

• Encodings: State, 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 directly 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

- **Algorithm Sketch**
  - 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

2 Methods:

1. Row Matching
2. Implication Chart
Implication Chart Method

- Sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State X=0</th>
<th>Next State X=1</th>
<th>Output X=0</th>
<th>Output X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3</td>
<td>S4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>S6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Implication Chart Method

Figure 1: Chart with entry of every pairs of states
Implication Chart Method
### Implication Chart Method

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>S1-S3</td>
<td>S2-S4</td>
<td>S1-S5</td>
<td>S2-S6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>S1-S0</td>
<td>S2-S0</td>
<td>S1-S0</td>
<td>S2-S0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>S1-S0</td>
<td>S2-S0</td>
<td>S1-S0</td>
<td>S2-S0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S0-S0</td>
<td>X</td>
</tr>
</tbody>
</table>

S0, S1, S2, S3, S5 $\rightarrow$ same output (group 1)
S4, S6 $\rightarrow$ same output (group 2) any combination across g1 and g2 indicate by X
Implied State Pairs

S0, S1, S2, S3, S5 → same output (group1)
S4, S6 → same output (group2) any combination across g1 and g2 indicate by X
## Implied State Pairs

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State X=0</th>
<th>Next State X=1</th>
<th>Output X=0</th>
<th>Output X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td><strong>S0</strong></td>
<td><strong>S1</strong></td>
<td><strong>S2</strong></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S2</td>
<td>S4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>S6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Rechecked Implied State Pairs

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S0-S0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>S0-S0</td>
<td>S0-S0</td>
<td>X</td>
</tr>
</tbody>
</table>

**IF square Si and Sj contains the ISP Sm-Sn and square Sm,Sn contains an X, then Mark Si,Sj with an X as well.**
Implication Chart Method

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset 0 + 1</td>
<td>S0</td>
<td>S1'</td>
<td>0</td>
</tr>
<tr>
<td>X0</td>
<td>S1'</td>
<td>S3'</td>
<td>0</td>
</tr>
<tr>
<td>X1</td>
<td>S3'</td>
<td>S4'</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>S0</td>
<td>S0</td>
<td>1</td>
</tr>
</tbody>
</table>

Simplified Table
Another Example

NurulHazlina/BEE2243/FSM
Divide into 2 groups;

A,B,C,E,F,G and D based on Output value
State Minimization Example

- Sequence Detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State X=0</th>
<th>Next State X=1</th>
<th>Output X=0</th>
<th>Output X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset 0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3</td>
<td>S4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>S6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Method of Successive Partitions

#### Row Matching

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>X=0</strong></td>
<td><strong>X=1</strong></td>
</tr>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>S6</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>S0</td>
</tr>
</tbody>
</table>

- (S0 S1 S2 S3 S4 S5 S6)  
  - S1 is equivalent to S2
- (S0 S1 S2 S3 S5) (S4 S6)  
  - S3 is equivalent to S5
- (S0 S3 S5) (S1 S2) (S4 S6)  
  - S4 is equivalent to S6
- (S0) (S3 S5) (S1 S2) (S4 S6)

**Input**

**Next State**

**Output**
Minimized FSM

- State minimized sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X=0</td>
<td>X=1</td>
</tr>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1'</td>
<td>0</td>
</tr>
<tr>
<td>0 + 1</td>
<td>S1'</td>
<td>S1'</td>
<td>0</td>
</tr>
<tr>
<td>X0</td>
<td>S3'</td>
<td>S4'</td>
<td>0</td>
</tr>
<tr>
<td>X1</td>
<td>S4'</td>
<td>S0</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram:

- States: S0, S1', S3', S4'
- Inputs: X, X/0
- Transitions:
  - S0 → S1' on X=0
  - S1' → S3' on X=0
  - S3' → S4' on X=0
  - S4' → S0 on X/0
  - S0 → S0 on X/0
  - S1' → S1' on X/0
  - S3' → S3' on X/0
  - S4' → S4' on X/0
Implication Chart
More Complex State Minimization

- Multiple input example

```
<table>
<thead>
<tr>
<th>Present State</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S3</td>
<td>S1</td>
<td>S4</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1</td>
<td>S0</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1</td>
<td>S4</td>
<td>S0</td>
<td>S5</td>
<td>0</td>
</tr>
</tbody>
</table>
```

inputs here

symbolic state transition table
Minimized FSM

- **Implication Chart Method**
  - Cross out incompatible states based on outputs
  - Then cross out more cells if indexed chart entries are already crossed out

<table>
<thead>
<tr>
<th>present state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0'</td>
<td>S0' S1 S2 S3'</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0' S3' S1 S3'</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1 S3' S2 S0'</td>
<td>1</td>
</tr>
<tr>
<td>S3'</td>
<td>S1 S0' S0' S3'</td>
<td>0</td>
</tr>
</tbody>
</table>

minimized state table (S0==S4) (S3==S5)
Minimizing Incompletely Specified FSMs

- Equivalence of states is transitive when machine is fully specified.
- But it's not transitive when don't cares are present.
  
<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>
</tbody>
</table>

  S1 is compatible with both S0 and S2, but S0 and S2 are incompatible.

- No polynomial time algorithm exists for determining the 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 input changes from 0 to 1

\[
\begin{array}{c|c|c}
X & Q_1 & Q_0 \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
- & 1 & 0 \\
\end{array}
\]

\[Q_{1^+} = X \ (Q_1 \ xor \ Q_0)\]

\[Q_{0^+} = X \ Q_1' \ Q_0'\]
Another Implementation of Edge Detector

- "Ad hoc" solution - not minimal but cheap and fast
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
Finite State Machine Concept

FSMs are the decision making logic of digital designs
Partitioning designs into datapath and control elements
When inputs are sampled and outputs asserted

Basic Design Approach: 4-step Design Process

Implementation Examples and Case Studies

- Finite-string pattern recognizer – mini project a
- Traffic light controller – mini project b
- Door combination lock – mini project c

SEQUENTIAL LOGIC EXAMPLES
General FSM Design Procedure

1. Determine inputs and outputs
2. Determine possible states of machine
   - State minimization
3. Encode states and outputs into a binary code
   - State assignment or state encoding
   - Output encoding
   - Possibly input encoding (if under our control)
4. Realize logic to implement functions for states and outputs
   - Combinational logic implementation and optimization
   - Choices in steps 2 and 3 have large effect on resulting logic
Finite String Pattern Recognizer (Step 1)

- **Finite String Pattern Recognizer**
  - One input (X) and one output (Z)
  - Output is asserted whenever the input sequence \( ...010... \) has been observed, as long as the sequence 100 has never been seen

- **Step 1: Understanding the Problem Statement**
  - Sample input/output behavior:

\[
\begin{align*}
  X: & \quad 0 0 1 0 1 0 1 0 0 1 0 \ldots \\
  Z: & \quad 0 0 0 1 0 1 0 1 0 0 0 \ldots \\

  X: & \quad 1 1 0 1 1 0 1 0 0 1 0 \ldots \\
  Z: & \quad 0 0 0 0 0 0 1 0 0 0 \ldots
\end{align*}
\]
Finite String Pattern Recognizer (Step 2)

- Step 2: Draw State Diagram
  - For the strings that must be recognized, i.e., 010 and 100
  - Moore implementation
Exit conditions from state S3: have recognized ...010
  – If next input is 0 then have ...0100 = ...100 (state S6)
  – If next input is 1 then have ...0101 = ...01 (state S2)

Exit conditions from S1:
recognizes strings of form ...0 (no 1 seen);
loop back to S1 if input is 0

Exit conditions from S4:
recognizes strings of form ...1 (no 0 seen);
loop back to S4 if input is 1
Finite String Pattern Recognizer (Step 2, cont’d)

- S2 and S5 still have incomplete transitions
  - S2 = ...01; If next input is 1, then string could be prefix of (01)1(00)
    - S4 handles just this case
  - S5 = ...10; If next input is 1, then string could be prefix of (10)1(0)
    - S2 handles just this case

- Reuse states as much as possible
  - Look for same meaning
  - State minimization leads to smaller number of bits to represent states

- Once all states have complete set of transitions we have final state diagram
Finite String Pattern Recognizer

- Review of Process
  - Understanding problem
    - Write down sample inputs and outputs to understand specification
  - Derive a state diagram
    - Write down sequences of states and transitions for sequences to be recognized
  - Minimize number of states
    - Add missing transitions; reuse states as much as possible
  - State assignment or encoding
    - Encode states with unique patterns
  - Simulate realization
    - Verify I/O behavior of your state diagram to ensure it matches specification
Traffic Light Controller as Two Communicating FSMs

- Without Separate Timer
  - $S_0$ would require 7 states
  - $S_1$ would require 3 states
  - $S_2$ would require 7 states
  - $S_3$ would require 3 states
  - $S_1$ and $S_3$ have simple transformation
  - $S_0$ and $S_2$ would require many more arcs
    - $C$ could change in any of seven states

- By Factoring Out Timer
  - Greatly reduce number of states
    - 4 instead of 20
  - Counter only requires seven or eight states
    - 12 total instead of 20
Communicating Finite State Machines

- One machine's output is another machine's input

machines advance in lock step
initial inputs/outputs: $X = 0, Y = 0$
Datapath and Control

- Digital hardware systems = data-path + control
  - Datapath: registers, counters, combinational functional units (e.g., ALU), communication (e.g., busses)
  - Control: FSM generating sequences of control signals that instructs datapath what to do next

status info and inputs -> state

control signal outputs -> control

control -> "puppeteer who pulls the strings"

data-path -> "puppet"
Digital Combinational Lock

- Door Combination Lock:
  - Punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - Inputs: sequence of input values, reset
  - Outputs: door open/close
  - Memory: must remember combination or always have it available
  - Open questions: how do you set the internal combination?
    - Stored in registers (how loaded?)
    - Hardwired via switches set by user
Determining Details of the Specification

- How many bits per input value?
- How many values in sequence?
- How do we know a new input value is entered?
- What are the states and state transitions of the system?
Digital Combination Lock State Diagram

- **States**: 5 states
  - Represent point in execution of machine
  - Each state has outputs
- **Transitions**: 6 from state to state, 5 self transitions, 1 global
  - Changes of state occur when clock says its ok
  - Based on value of inputs
- **Inputs**: reset, new, results of comparisons
- **Output**: open/closed

![Digital Combination Lock State Diagram](image-url)
Datapath and Control Structure

• Datapath
  - Storage registers for combination values
  - Multiplexer
  - Comparator

• Control
  - Finite-state machine controller
  - Control for data-path (which value to compare)
State Table for Combination Lock

- Finite-State Machine
  - Refine state diagram to take internal structure into account
  - State table ready for encoding

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S1</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S1</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S3</td>
<td>OPEN</td>
<td>-</td>
<td>open</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Encodings for Combination Lock

- Encode state table
  - State can be: S1, S2, S3, OPEN, or ERR
    - Needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - And as many as 5: 00001, 00010, 00100, 01000, 10000
    - Choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - Output mux can be: C1, C2, or C3
    - Needs 2 to 3 bits to encode
    - Choose 3 bits: 001, 010, 100
  - Output open/closed can be: open or closed
    - Needs 1 or 2 bits to encode
    - Choose 1 bit: 1, 0

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>–</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>0000</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>1000</td>
<td>–</td>
<td>1</td>
</tr>
</tbody>
</table>

mux is identical to last 3 bits of state, open/closed is identical to first bit of state, therefore, we do not even need to implement FFs to hold state, just use outputs.
Datapath Implementation for Combination Lock

- **Multiplexer**
  - Easy to implement as combinational logic when few inputs
  - Logic can easily get too big for most PLDs
Datapath Implementation (cont’d)

- **Tri-State Logic**
  - Utilize a third output state: “no connection” or “float”
  - Connect outputs together as long as only one is “enabled”
  - Open-collector gates can only output 0, not 1
    - Can be used to implement logical AND with only wires
Digital Combination Lock (New Datapath)

- Decrease number of inputs
- Remove 3 code digits as inputs
  - Use code registers
  - Make them loadable from value
  - Need 3 load signal inputs (net gain in input \(4 \times 3 - 3 = 9\))
    - Could be done with 2 signals and decoder (ld1, ld2, ld3, load none)
Section Summary

• FSM Design
  – Understanding the problem
  – Generating state diagram
  – Implementation using synthesis tools
  – Iteration on design/specification to improve qualities of mapping
  – Communicating state machines

• Four case studies
  – Understand I/O behavior
  – Draw diagrams
  – Enumerate states for the "goal"
  – Expand with error conditions
  – Reuse states whenever possible
FSM AND HDL
Hardware Description Languages and Sequential Logic

• Flip-flops
  – representation of clocks - timing of state changes
  – asynchronous vs. synchronous

• FSMs
  – structural view (FFs separate from combinational logic)
  – behavioral view (synthesis of sequencers – not in this course)

• Data-paths = data computation (e.g., ALUs, comparators) + registers
  – use of arithmetic/logical operators
  – control of storage elements
Example: reduce-1-string-by-1

- Remove one 1 from every string of 1s on the input
Verilog FSM - Reduce 1s example

- **Moore machine**

```verilog
module reduce (clk, reset, in, out);
    input clk, reset, in;
    output out;

    parameter zero  = 2'b00;
    parameter one1  = 2'b01;
    parameter two1s = 2'b10;

    reg out;
    reg [2:1] state;    // state variables
    reg [2:1] next_state;

    always @(posedge clk)
        if (reset) state = zero;
        else       state = next_state;
```

state assignment
(easy to change, if in one place)
always @(in or state)

    case (state)
        zero:
            // last input was a zero
            begin
                if (in) next_state = one1;
                else    next_state = zero;
            end
        one1:
            // we've seen one 1
            begin
                if (in) next_state = two1s;
                else    next_state = zero;
            end
        two1s:
            // we've seen at least 2 ones
            begin
                if (in) next_state = two1s;
                else    next_state = zero;
            end
    endcase

endmodule
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables
  reg next_state;

  always @(posedge clk)
    if (reset) state = zero;
    else       state = next_state;

  always @(in or state)
    case (state)
      zero:    // last input was a zero
        begin
          out = 0;
          if (in) next_state = one;
          else    next_state = zero;
        end
      one:     // we've seen one 1
        if (in) begin
          next_state = one; out = 1;
        end else begin
          next_state = zero; out = 0;
        end
    endcase
endmodule
Synchronous Mealy Machine

module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables
  always @(posedge clk)
    if (reset) state = zero;
    else
      case (state)
        zero:       // last input was a zero
          begin
            out = 0;
            if (in) state = one;
            else    state = zero;
          end
        one:       // we've seen one 1
          if (in) begin
            state = one; out = 1;
          end else begin
            state = zero; out = 0;
          end
      endcase
endmodule