Abstraction of State Elements

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

![Abstraction Diagram]

Inputs  →  Combinational Logic  →  Outputs

State Inputs  →  Storage Elements  →  State Outputs
Finite State Machine Representation

- 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

Pieces of FSMs

- **NSD** (Next State Decoder)
  - Inputs
  - Present state
  - Outputs
  - New value for next state

- **OD** (Output Decoder)
  - Inputs
  - Present state
  - Outputs
  - Next state to load on rising edge

- **State register**
  - Clock
  - Present state

Diagram showing transitions between states and connections to state register.
Types of FSM’s

Moore and Mealey FSM

Moore

- Inputs: NSD
- Outputs: OD
- Function only of present state

Mealey

- Inputs: NSD, NS
- Outputs: OD

Example Finite State Machine

- Combination lock from first lecture

Example diagram with states and transitions:
Shift Registers as FSM’s

- Shift Register
  - Input value shown on transition arcs
  - Output values shown within state node

Counters as FSM’s

- Counters
  - Proceed thru well-defined state sequence 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, ...
Turn a State Diagram into Logic

- Counter
  - Three 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

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

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

- Programmable Logic Building Block for Sequential Logic
- Macro-cell: FF + logic
  - D-FF
  - Two-level logic capability like PAL (e.g., 8 product terms)

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

N1 := In
N2 := C1
N3 := C2

\[
\text{In} \quad \text{C1} \quad \text{C2} \quad \text{C3} \quad \text{N1} \quad \text{N2} \quad \text{N3}
\]

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

N1 := In
N2 := C1
N3 := C2

<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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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: S1, S2, ..., Sk
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: Fs(Si, Ij)
- Output function: Fo(Si) or Fo(Si, Ij)
Specifying Outputs: Moore Machine

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

```
reset  current  next
input  state  state  output
1  -  -  A
0  0  A  B  0
0  1  A  C  0
0  0  B  B  0
0  1  B  D  0
0  0  C  E  0
0  1  C  C  0
0  0  D  E  1
0  1  D  C  1
0  0  E  B  1
0  1  E  D  1
```

Specifying Outputs: Mealy Machine

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

```
reset  current  next
input  state  state  output
1  -  -  A
0  0  A  B  0
0  1  A  C  0
0  0  B  B  0
0  1  B  C  1
0  0  C  B  1
0  1  C  C  0
```
Comparison: Mealy/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
- 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

Mealy and Moore Examples

- Recognize \(A, B = 0, 1\)
  - Mealy or Moore?
Mealy and Moore Examples

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

Registered Mealy (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
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 0 0</td>
<td>0</td>
<td>0</td>
<td>LOST F</td>
<td></td>
</tr>
<tr>
<td>LOST - 1</td>
<td>E/G F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOST 1 -</td>
<td>E/G F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 0 0</td>
<td>B TL, F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 0 1</td>
<td>A TL, F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 1 -</td>
<td>E/G TL, F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B - 0</td>
<td>C TR, F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B - 1</td>
<td>A TR, F</td>
<td></td>
<td></td>
<td></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</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOST 0 0 0</td>
<td>0 0 0</td>
<td>F TR TL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOST - 1 0 1</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOST 1 - 0 1</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 0 0 1</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 0 1 0</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 1 - 0 1</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B - 0 1 0</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B - 1 0 1</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>... ...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It now remains to synthesize these 6 functions.
Synthesis - State, Output Functions

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

e.g.

TR = X + Y Z
X' = X R' + Y Z R' = R' TR

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

State Assignment

- Choose bit vectors to assign to each "symbolic" state
  - With n state bits for m states there are $2^n! / (2^n - m)!$
    - $[\log n <= m <= 2^n]$ 
  - $2^n$ codes possible for 1st state, $2^n-1$ for 2nd, $2^n-2$ for 3rd, ...
  - Huge number even for small values of n and m
    - Intractable for state machines of any size
    - Heuristics are necessary for practical solutions
  - Optimize some metric for the combinational logic
    - Size (amount of logic and number of FFs)
    - Speed (depth of logic and fanout)
    - Dependencies (decomposition)
State Assignment Strategies

- Possible Strategies
  - Sequential – just number states as they appear in the state table
  - Random – pick random codes
  - One-hot – use as many state bits as there are states (bit=1 -> state)
  - 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, debug
- 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 its dependent upon)
- Impractical for Large Machines
  - Too many states require too many flip-flops
  - Decompose FSMs into smaller pieces that can be one-hot encoded
- Many Slight Variations to One-hot
  - One-hot + all-0
Heuristics for State Assignment

- Adjacent codes to states that share a common next state
  - Group 1's in next state map
    \[
    \begin{array}{c|ccc}
    I & Q & Q^+ & O \\
    \hline
    i & a & c & j \\
    b & c & k \\
    \end{array}
    \]
  - Adjacent codes to states that share a common ancestor state
    - Group 1's in next state map
    \[
    \begin{array}{c|ccc}
    I & Q & Q^+ & O \\
    \hline
    i & a & b & j \\
    k & a & c & l \\
    \end{array}
    \]
  - Adjacent codes to states that have a common output behavior
    - Group 1's in output map
    \[
    \begin{array}{c|ccc}
    I & Q & Q^+ & O \\
    \hline
    i & a & b & j \\
    c & d & j \\
    \end{array}
    \]

Heuristic State Assignment

- All current methods are variants of this
  1) Determine which states “attract” each other (weighted pairs)
  2) Generate constraints on codes (which should be in same cube)
  3) Place codes on Boolean cube so as to maximize 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
- Why create new functions for state bits when output can serve as well
- Fits in nicely with synchronous Mealy implementations

HG = ST' H1' H0' F1 F0' + ST H1 H0' F1 F0
HY = ST H1' H0 F1 F0' + ST' H1 H0 F1 F0'
FG = ST H1' H0 F1 F0' + ST' H1 H0' F1 F0'
HY = ST H1 H0' F1' F0' + ST' H1 H0' F1' F0

Output patterns are unique to states, we do not need ANY state bits – implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Present State</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C TL TS</td>
<td>ST H F</td>
<td>ST H F</td>
<td>ST H F</td>
</tr>
<tr>
<td>0 0 0</td>
<td>HG HG</td>
<td>0 0 10</td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>HG HY</td>
<td>1 0 10</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>HY HG</td>
<td>0 1 10</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>FG FG</td>
<td>0 1 00</td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>FG FG</td>
<td>1 1 00</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>HG HG</td>
<td>1 1 01</td>
<td></td>
</tr>
</tbody>
</table>

Current State Assignment

- For tight 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
  - Ad hoc - no tools
  - Most common approach taken by human designers
  - Yields very small circuits for most FSMs
Example: Vending Machine

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

Example: Vending Machine

- 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

- Minimize number of states - reuse states whenever possible

```
state  present inputs next output
D N
0¢  0  0  0¢  0
1  1  10¢  0
1  1  15¢  1
5¢  0  0  5¢  0
1  1  10¢  0
1  1  15¢  0
10¢  0  0  10¢  0
1  1  15¢  0
1  1  15¢  0
15¢  0  0  15¢  1

symbolic state table
```

Example: Vending Machine

- Uniquely Encode States

```
present state  inputs next state output
Q1 Q0 D N D1 D0 open
0 0 0 0 0 0
0 1 0 1 0
1 0 1 0 0
1 1 1 1 1
0 1 0 1 0
1 0 1 1 0
1 1 1 1 1
1 0 1 1 0
1 1 1 1 1
1 1 1 1 1
```

EECS150 - Fall 2001
Example: Vending Machine

- Mapping to Logic

\[ D_1 = Q_1 + D + Q_0 \cdot N \]
\[ D_0 = Q_0' \cdot N + Q_0 \cdot N' + Q_1 \cdot N + Q_1 \cdot D \]
\[ OPEN = Q_1 \cdot Q_0 \]

PLD mapping

\[ D_0 = \text{reset}'(Q_0' \cdot N + Q_0 \cdot N' + Q_1 \cdot N + Q_1 \cdot D) \]
\[ D_1 = \text{reset}'(Q_1 + D + Q_0 \cdot N) \]
\[ OPEN = Q_1 \cdot Q_0 \]
Vending Machine

- 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.

Retimed PLD Mapping
Example: Vending Machine

One-hot Encoding

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>next state output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3, Q2, Q1, Q0, D, N</td>
<td>D0, D1, D2, D3, open</td>
<td>D0 = Q0 D’ N’</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0</td>
<td>0 0 0 1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>- - - -</td>
<td></td>
</tr>
</tbody>
</table>

| 0 0 1 0 | 0 0 | 0 0 1 0 0 |
| 0 1 | 0 1 0 0 0 |
| 1 0 | 1 0 0 0 0 |
| 1 1 | - - - - |

| 0 1 0 0 | 0 0 | 0 1 0 0 0 |
| 0 1 | 1 0 0 0 0 |
| 1 0 | 1 0 0 0 0 |
| 1 1 | - - - - |

| 1 0 0 0 | - - | 1 0 0 0 1 |

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
- 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

- Highway/farm road intersection

![Diagram of highway/farm road intersection with car sensors]

<table>
<thead>
<tr>
<th>Tabulation of Inputs and Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>inputs</strong></td>
</tr>
<tr>
<td>reset</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>TS</td>
</tr>
<tr>
<td>TL</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

- **State Diagram**

  ![State Diagram](image)

  - S0: HG
  - S1: HY
  - S2: FG
  - S3: FY

Example: Traffic Light Controller

- 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>C TL TS</td>
<td>HG</td>
<td>SG</td>
<td>ST H F</td>
</tr>
<tr>
<td>0 0 0</td>
<td>HG</td>
<td>HG</td>
<td>0 Green Red</td>
</tr>
<tr>
<td>0 0 1</td>
<td>HG</td>
<td>HG</td>
<td>0 Green Red</td>
</tr>
<tr>
<td>0 1 0</td>
<td>HG</td>
<td>HY</td>
<td>0 Yellow Red</td>
</tr>
<tr>
<td>0 1 1</td>
<td>HG</td>
<td>HY</td>
<td>0 Yellow Red</td>
</tr>
<tr>
<td>1 0 0</td>
<td>FG</td>
<td>FG</td>
<td>0 Red Green</td>
</tr>
<tr>
<td>1 0 1</td>
<td>FG</td>
<td>FY</td>
<td>0 Red Green</td>
</tr>
<tr>
<td>1 1 0</td>
<td>FY</td>
<td>FY</td>
<td>0 Red Yellow</td>
</tr>
<tr>
<td>1 1 1</td>
<td>FY</td>
<td>HG</td>
<td>0 Red Yellow</td>
</tr>
</tbody>
</table>

- **State Assignments**
  - 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 State Assignments

## SA1

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{NS}_1$</td>
<td>$\text{NS}_0$</td>
<td>$\text{ST}$</td>
<td>$\text{H}_1$</td>
<td>$\text{F}_1$</td>
<td>$\text{H}_0$</td>
</tr>
<tr>
<td>$C\cdot TL\cdot PS_1\cdot PS_0 \lor TS\cdot PS_1\cdot PS_0 + TS\cdot PS_1\cdot PS_0' \lor C\cdot PS_1\cdot PS_0 \lor TL\cdot PS_1\cdot PS_0$</td>
<td>$C\cdot TL\cdot PS_1\cdot PS_0' \lor C\cdot TL\cdot PS_1\cdot PS_0 + PS_1\cdot PS_0$</td>
<td>$C\cdot TL\cdot PS_1\cdot PS_0' \lor TS\cdot PS_1\cdot PS_0 + TS\cdot PS_1\cdot PS_0' \lor C\cdot PS_1\cdot PS_0 \lor TL\cdot PS_1\cdot PS_0$</td>
<td>$PS_1$</td>
<td>$PS_1' \lor PS_0$</td>
<td>$PS_1' \lor PS_0$</td>
</tr>
</tbody>
</table>

## SA2

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{NS}_1$</td>
<td>$\text{NS}_0$</td>
<td>$\text{ST}$</td>
<td>$\text{H}_1$</td>
<td>$\text{F}_1$</td>
<td>$\text{H}_0$</td>
</tr>
<tr>
<td>$C\cdot TL\cdot PS_1$ + $TS\cdot PS_1 + C\cdot PS_1\cdot PS_0$</td>
<td>$TS\cdot PS_1 + C\cdot PS_1\cdot PS_0$</td>
<td>$C\cdot TL\cdot PS_1 + C\cdot PS_1\cdot PS_0 + TS\cdot PS_1$</td>
<td>$PS_0$</td>
<td>$PS_0$</td>
<td>$PS_1\cdot PS_0$</td>
</tr>
</tbody>
</table>

## SA3

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{NS}_3$</td>
<td>$\text{NS}_2$</td>
<td>$\text{NS}_1$</td>
<td>$\text{ST}$</td>
<td>$\text{H}_1$</td>
<td>$\text{F}_1$</td>
</tr>
<tr>
<td>$C\cdot PS_2 + TL\cdot PS_2 + TS\cdot PS_3$</td>
<td>$TS\cdot PS_1 + C\cdot TL\cdot PS_2$</td>
<td>$C\cdot PS_2 + TL\cdot PS_2 + TS\cdot PS_3$</td>
<td>$C\cdot PS_2 + TL\cdot PS_2 + TS\cdot PS_3$</td>
<td>$PS_3 + PS_2$</td>
<td>$PS_1 \lor PS_0$</td>
</tr>
</tbody>
</table>