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

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

Example Finite State Machine

- Combination lock from first lecture
Shift Registers as FSM’s

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

![Shift Register Diagram](image)

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

![State Transition Table Diagram](image)

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, ...

![Counter Diagram](image)

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)

![Implementation Diagram](image)

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

![Counter Diagram](image)

Another Example

- Shift Register
  - Input determines next state

![Another Example Diagram](image)
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)

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

Specifying Outputs: Moore Machine

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

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
- LOST: Forward until we touch something
- G: Turn left until...

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
- 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</td>
<td>0</td>
<td>0</td>
<td>LOST</td>
<td>F</td>
</tr>
<tr>
<td>LOST 1</td>
<td>0</td>
<td>1</td>
<td>E/G</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>TR</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>TR</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>TR</td>
<td>F</td>
</tr>
</tbody>
</table>

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

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>000</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>1 0 0</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>1 0 0</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>1 0 1</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>1 0 1</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>1 1 0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>1 1 0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>1 1 0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

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

**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

**State Assignment**

- Choose bit vectors to assign to each “symbolic” state
  - With n state bits for m states there are \(2^n \div (2^n - m)!\)
  - \(2^n\) 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)

**Heuristics for State Assignment**

- Adjacent codes to states that share a common next state
  - Group 1’s in next state map

**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

**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 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

Input 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)

Inputs Present State Next State Outputs
CT L T S S T H F
0–– H G H G 00 0 1 0
–0– H G H G 00 0 1 0
11– H G H Y 10 0 1 0
––0 H Y H Y 00 1 1 0
––1 H Y F G 10 1 1 0
1 0 – FG FG 0 10 00
0 – – FG FY 1 10 00
– 1 – FG FY 1 10 00
– – 0 FY FY 0 10 01
– – 1 FY HG 1 10 01

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

- 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)

Minimize number of states - reuse states whenever possible

Example: Vending Machine

- Uniquely Encode States
Example: Vending Machine

- Mapping to Logic

\[
D_0 = \text{reset}'(Q_0'N + Q_0N' + Q_1N + Q_1D) \\
D_1 = \text{reset}'(Q_1 + D + Q_0N) \\
\text{OPEN} = Q_1 Q_0
\]

- One-hot Encoding

<table>
<thead>
<tr>
<th>present state</th>
<th>input</th>
<th>next state output</th>
<th>open</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>0 1</td>
<td>0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1 0</td>
<td>0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0</td>
<td>1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 ---- ----</td>
<td>1 0</td>
<td>1 1 ----</td>
<td>0</td>
</tr>
</tbody>
</table>

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)
- Implementation now looks like a synchronous Mealy machine.
- Common for programmable devices to have FF at end of logic.

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

```
farm road

car sensors

Highway
```

Example: Traffic Light Controller

- 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>FG, FY, FR</td>
<td>assert green/yellow/red highway lights</td>
</tr>
<tr>
<td>C</td>
<td>detect vehicle on farm rd</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/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

- Logic for State Assignments

```
SA1: NS1 = C•TL•PS1 + C•TL•PS1 + TS•PS1 + TS•PS1 + C'•PS1 + TL•PS1
NS0 = TS•PS1 + PS1 + TS•PS1 + TS•PS1 + C'•PS1 + TL•PS1
ST = C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1
H1 = PS1 F1 = PS1
F1 = PS1
SA2: NS1 = C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1
NS0 = TS•PS1 + PS1 + TS•PS1 + TS•PS1 + C'•PS1 + TL•PS1
ST = C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1
H1 = PS1 F1 = PS1
F1 = PS1
SA3: NS1 = C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1
NS0 = PS1 + TS•PS1 + TS•PS1 + TS•PS1 + TS•PS1 + TS•PS1
ST = C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1 + C•TL•PS1
H1 = PS1 F1 = PS1
F1 = PS1
```

Example: Traffic Light Controller

- State Diagram