6. Finite State Machines

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS
Our New Machine

- Engineered cycles
- Works only if dynamic discipline obeyed
- Remembers $k$ bits for a total of $2^k$ unique combinations

- Acyclic graph
- Obeys static discipline
- Can be exhaustively enumerated by a truth table of $2^{k+m}$ rows and $k+n$ output columns
A Simple Sequential Circuit...

Let's make a digital binary *Combination Lock*:

**Specification:**

- One input ("0" or "1")
- One output ("Unlock" signal)
- UNLOCK is 1 if and only if:
  Last 4 inputs were the "combination": 0110
A FINITE STATE MACHINE has

- $k$ STATES: $S_1 \ldots S_k$ (one is “initial” state)
- $m$ INPUTS: $I_1 \ldots I_m$
- $n$ OUTPUTS: $O_1 \ldots O_n$
- Transition Rules: $s'(s, I)$ for each state $s$ and input $I$
- Output Rules: $\text{Out}(s)$ or $\text{Out}(s, I)$ for each state $s$ and input $I$
Designing our lock ...

- Need an initial state; call it SX.
- Must have a separate state for each step of the proper entry sequence
- Must handle other (erroneous) entries
Arrows leaving a state must be:

1. **mutually exclusive**
   - can’t have two choices for a given input value

2. **collectively exhaustive**
   - every state must specify what happens for each possible input combination. “Nothing happens” means arc back to itself.
State Transition Diagram as a Truth Table

All state transition diagrams can be described by truth tables...

Binary encodings are assigned to each state (a bit of an art)

The truth table can then be simplified using the reduction techniques we learned for combinational logic

<table>
<thead>
<tr>
<th>IN Current State</th>
<th>Next State Unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000 SX</td>
<td>S0 001 0</td>
</tr>
<tr>
<td>1 000 SX</td>
<td>SX 000 0</td>
</tr>
<tr>
<td>0 001 S0</td>
<td>S0 001 0</td>
</tr>
<tr>
<td>1 001 S0</td>
<td>S01 011 0</td>
</tr>
<tr>
<td>0 011 S01</td>
<td>S0 001 0</td>
</tr>
<tr>
<td>1 011 S01</td>
<td>S01 010 0</td>
</tr>
<tr>
<td>0 010 S011</td>
<td>S0110 100 0</td>
</tr>
<tr>
<td>1 010 S011</td>
<td>SX 000 0</td>
</tr>
<tr>
<td>0 100 S0110</td>
<td>S0 001 1</td>
</tr>
<tr>
<td>1 100 S0110</td>
<td>S01 011 1</td>
</tr>
</tbody>
</table>

The assignment of codes to states can be arbitrary, however, if you choose them carefully you can greatly reduce your logic requirements.
We assume inputs are synchronized with clock...

4 inputs \( \Rightarrow 2^4 \) locations, each location supplies 4 bits

5 states \( \Rightarrow 3\)-bit encoding

Trigger update periodically ("clock")
Discrete State, Discrete Time

Two design choices:
1. outputs only depend on state (Moore)
2. outputs depend on inputs + state (Mealy)

$s$ state bits $\Rightarrow$ $2^s$ possible states
Housekeeping Issues...

1. Initialization? Clear the memory?
2. Unused state encodings?
   - waste ROM (use gates)
   - what does it mean?
   - can the FSM recover?
3. Choosing encoding for state?
4. Synchronizing input changes with state update?

Now, that’s a funny looking state machine
1. What can you say about the number of states?

2. Same question:

3. Here’s an FSM. Can you discover its rules?
What’s My Transition Diagram?

0=OFF, 1=ON?
"1111" = 0
Surprise!

- If you know NOTHING about the FSM, you’re never sure!
- If you have a BOUND on the number of states, you can discover its behavior:

  K-state FSM: Every (reachable) state can be reached in < k steps.

BUT ... FSMs may be equivalent!
**FSM Equivalence**

ARE THEY DIFFERENT?
NOT in any practical sense! They are EXTERNALLY INDISTINGUISHERIBLE, hence interchangeable.

FSMs are *EQUIVALENT* iff every input sequence yields identical output sequences.

ENGINEERING GOAL:
- HAVE an FSM which *works*...
- WANT simplest (ergo cheapest) equivalent FSM.
Let’s Build a RoboAnt

- **SENSORS**: antennae L and R, each 1 if in contact with something.
- **ACTUATORS**: Forward Step F, ten-degree turns TL and TR (left, right).

**GOAL**: Make our ant smart enough to get out of a maze like:

![Maze Diagram](image)

**STRATEGY**: "Right antenna to the wall"
Lost In Space

Action: Go forward until we hit something.

“lost” is the initial state
Bonk!

Action: Turn left (CCW) until we don’t touch anymore
A Little to the Right...

Action: Step and turn right a little, look for wall
Then a Little to the Left

Action: Step and turn left a little, till not touching (again)
Dealing With Outside Corners

Action: Step and turn right until we hit perpendicular wall

Hey, this might even work!
Observation: two states are equivalent if
1. Both states have identical outputs; **AND**
2. Every input $\Rightarrow$ equivalent states.

Reduction Strategy:
Find pairs of equivalent states, MERGE them.
Merge equivalent states Wall1 and Corner into a single new, combined state.

Behaves exactly as previous (5-state) FSM, but requires half the ROM in its implementation!
Building The Transition Table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>00 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 0 1</td>
<td>01 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 0</td>
<td>01 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 1</td>
<td>01 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0 1</td>
<td>01 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 0</td>
<td>01 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 1</td>
<td>01 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LOST → RCCW

**Transition Table**

<table>
<thead>
<tr>
<th>S</th>
<th>L</th>
<th>R</th>
<th>S'</th>
<th>TR</th>
<th>TL</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>00 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 0 1</td>
<td>01 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 0</td>
<td>01 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 1 1</td>
<td>01 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0 1</td>
<td>01 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 0</td>
<td>01 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 1</td>
<td>01 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Implementation Details

<table>
<thead>
<tr>
<th>S</th>
<th>L</th>
<th>R</th>
<th>S'</th>
<th>TR</th>
<th>TL</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>-</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>-</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>-</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Complete Transition table**

\[
S_0' = R + \overline{LS_1} + LS_0
\]

\[
S_1' = S_1\overline{S_0} + \overline{LS_1} + \overline{LRS_0}
\]
Ant Schematic
FSMs All the Way Down?

• More than ants:
  Swarming, flocking, and schooling can result from collections of very simple FSMs

• Perhaps most physics:
  Cellular automata, arrays of simple FSMs, can more accurately model fluids than numerical solutions to PDEs

• What if:
  We replaced the ROM with a RAM and have outputs that modify the RAM?

... You'll see FSMs for the rest of your life!
The World Doesn’t Run on Our Clock!

What if each button input is an asynchronous 0/1 level?

To build a system with asynchronous inputs, we have to break the rules: *we cannot guarantee that setup and hold time requirements are met at the inputs!*

So, we need a “synchronizer” at each input:

*Valid except for brief periods following active clock edges*
The Bounded-time Synchronizer

A classic problem

UNSOLVABLE

For NO finite values of $t_E$ and $t_D$ is this spec realizable, even with reliable components!

Synchronizer specifications:
- finite $t_D$ (decision time)
- finite $t_E$ (allowable error)
- value of $S$ at time $t_C + t_D$:
  - 1 if $t_{IN} < t_C - t_E$
  - 0 if $t_{IN} > t_C + t_E$
  - 0, 1 otherwise

CASE 1
CASE 2
CASE 3
Let’s just use a D Register:

\[ \text{IN: } \_ \_ \text{ at } t_{IN} \rightarrow \text{D } Q \]

\[ \text{CLK: } \_ \_ \text{ at } t_{C} \rightarrow \]

We’re lured by the digital abstraction into assuming that Q must be either 1 or 0. But let’s look at the input latch in the flip flop when IN and CLK change at about the same time...

**DECISION TIME** is \( T_{PD} \) of register.

**ALLOWABLE ERROR** is \( \max(t_{SETUP}, t_{HOLD}) \)

Our logic:

\( T_{PD} \) after \( T_{C} \), we’ll have

- \( Q=1 \iff t_{IN} + t_{SETUP} < t_{C} \)
- \( Q=0 \iff t_{C} + t_{HOLD} < t_{IN} \)
- \( Q=0 \) or 1 otherwise.
The Mysterious Metastable State

Recall that the latch output is the solution to two simultaneous constraints:

1. The VTC of path thru MUX; and
2. $V_{in} = V_{out}$

In addition to our expected stable solutions, we find an unstable equilibrium in the forbidden zone called the “Metastable State”
Metastable State: Properties

1. It corresponds to an *invalid* logic level.
2. It’s an *unstable* equilibrium; a small perturbation will cause it to move toward a stable 0 or 1.
3. It will settle to a valid 0 or 1... eventually.
4. BUT – depending on how close it is to the $V_{in} = V_{out}$ “fixed point” of the device – it may take arbitrarily long to settle out.
5. EVERY bistable system exhibits at least one metastable state!

If metastable at $t_0$:

- $p(\text{metastable at } t_0 + T) > 0$ for finite $T$
- $p(\text{metastable at } t_0 + T)$ decreases exponentially with increasing $T$
Solution: Delay Increases Reliability

Extra registers between the asynchronous input and your logic are the best insurance against metastable states.

For higher clock rates, consider adding additional registers.

Quarantine time reduces $p(\text{metastable})$