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

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

- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)

Finite state machine representations

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements

Sequential logic
- sequences through a series of states
- based on sequence of values on input signals
- clock period defines elements of sequence
Example finite state machine diagram

- Combination lock from introduction to course
  - 5 states
  - 5 self-transitions + 1 reset to state S1
  - 6 transitions + 4 resets

Can any sequential system be represented with a state diagram?

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

#### 3. Start with counters
- simple because output is just state
- simple because no choice of next state based on input

#### 3. State diagram to state transition table
- tabular form of state diagram
- like a truth-table

#### 3. State encoding
- decide on representation of states
- for counters it is simple: just its value

#### 3. Implementation
- flip-flop for each state bit
- combinational logic based on encoding

### FSM design procedure: state diagram to encoded state transition table

#### 3. Tabular form of state diagram
- Like a truth-table (specify output for all input combinations)

#### 3. Encoding of states: easy for counters – just use value

![State Diagram]

<table>
<thead>
<tr>
<th>current 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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>C2</td>
<td>C1</td>
<td>N3</td>
<td>N2</td>
<td>N1</td>
</tr>
<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 <= C1'
N2 <= C1'C2
N3 <= C1'C3 + C1''C3
N1 <= (C1'C3' + (C1'C2')C3
N1 <= (C1'C2' xor C3

Back to the 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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Winter 2001
CSER370 · VI · Finite State Machines

Winter 2001
CSER370 · VI · Finite State Machines
More complex counter example

- **Complex counter**
  - Repeats 5 states in sequence
  - Not a binary number representation

- **Step 1:** Derive the state transition diagram
  - Count sequence: 000, 010, 011, 101, 110

- **Step 2:** Derive the state transition table from the state transition diagram

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

Note the don't care conditions that arise from the unused state codes.

More complex counter example (cont'd)

- **Step 3:** K-maps for next state functions

```
A  B  C+  X
0  0  0  0
0  0  1
0  1  0
0  1  1
1  0  0
1  0  1
1  1  0
1  1  1

A  B  C+  X
0  1  0  0
0  1  1
0  0  0
0  0  1
1  0  0
1  0  1
1  1  0
1  1  1
```

\[
C^+ = A \\
B^+ = B' + A'C \\
A^+ = BC'
\]
**Self-starting counters (cont'd)**

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

<table>
<thead>
<tr>
<th>C+</th>
<th>C</th>
<th>B+</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C+</th>
<th>C</th>
<th>B+</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

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

36 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: S1, S2, ..., Sk
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: F(s_i, I)
- Output function: F_o(s_i) or F_o(s_i, I)

```
  +-----+   +-----+   +-----+   +-----+   +-----+   +-----+
  |     |   |     |   |     |   |     |   |     |   |     |
  |     |   |     |   |     |   |     |   |     |   |     |
  +-----+   +-----+   +-----+   +-----+   +-----+   +-----+
```

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...

---

### Designing an ant brain

#### State diagram

![State diagram](image-url)

- **LOST** (F)
- **E,G** (TL)
- **A** (TL, F)
- **B** (TR, F)
- **C** (TR, F)

Winter 2001
**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</td>
<td>0</td>
<td>0</td>
<td>LOST F</td>
<td></td>
</tr>
<tr>
<td>LOST 1</td>
<td>1</td>
<td>-</td>
<td>E/G F</td>
<td></td>
</tr>
<tr>
<td>LOST 1</td>
<td>-</td>
<td>1</td>
<td>E/G F</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>T/L F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>T/L F</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>-</td>
<td>E/G F</td>
<td>T/L F</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>C</td>
<td>TR F</td>
</tr>
<tr>
<td>B</td>
<td>1</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</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X,Y,Z</td>
<td></td>
<td></td>
<td>X*, Y*, Z*</td>
<td>F TR TL</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td>0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td></td>
<td></td>
<td>0 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td></td>
<td></td>
<td>0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td></td>
<td></td>
<td>0 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td></td>
<td></td>
<td>0 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 1</td>
<td></td>
<td></td>
<td>0 1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 0</td>
<td></td>
<td></td>
<td>0 1 0 1 0</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 of next state and output functions

<table>
<thead>
<tr>
<th>state inputs</th>
<th>next state outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X,Y,Z 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 1 1 0 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 1 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 1 0 0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 0 1 1 0</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1 0 1 1 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1 0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 0 1 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1 0</td>
</tr>
</tbody>
</table>

E.g.

\[ TR = X + Y Z \]
\[ X^* = X R^* + Y Z R^* = R^* TR \]
**Circuit implementation**

36 Outputs are a function of the current state only - Moore machine

![Diagram of a Moore machine with output logic and next state logic](image)

**Don’t cares in FSM synthesis**

36 What happens to the "unused" states (101, 110, 111)?

36 They were exploited as don’t cares to minimize the logic

- if the states can’t happen, then we don’t care what the functions do
- if states do happen, we may be in trouble

![Diagram of FSM with don’t cares](image)
**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 inputs</th>
<th>next state outputs</th>
<th>X*</th>
<th>X</th>
<th>Y*</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, L, R</td>
<td>X', Y', F, TR, TL</td>
<td>L</td>
<td>R</td>
<td>L</td>
<td>R</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>----</td>
<td>---</td>
<td>----</td>
<td>---</td>
</tr>
<tr>
<td>00 0 0</td>
<td>00 1 0 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00 1 1</td>
<td>01 1 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01 0 0</td>
<td>11 0 0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01 1 1</td>
<td>01 0 0 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10 0 0</td>
<td>11 1 0 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10 1 1</td>
<td>10 1 0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11 0 0</td>
<td>11 1 0 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11 1 1</td>
<td>10 1 0 0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Mealy vs. Moore machines

- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our ant brain is a Moore machine
  - output does not react immediately to input change
- We could have specified a Mealy FSM
  - outputs have immediate reaction to inputs
  - as inputs change, so does next state, doesn’t commit until clocking event

Specifying 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

<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>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>A</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>D</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>D</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>
Specifying 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>A</td>
<td>0</td>
<td></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>

Comparison of Mealy and 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

36 Recognize A, B = 0,1
Mealy or Moore?

Mealy and Moore examples (cont'd)

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

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
define zero 0
define one1 1
define two1s 2

module reduce (clk, reset, in, out);
    input clk, reset, in;
    output out;
    reg out;
    reg [2:1] state;    // state variables
    reg [2:1] next_state;

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

**Mealy**
Moore Verilog FSM (cont’d)

```verilog
case #state
    'zero:
        // last input was a zero
        begin
            if #in then
                next_state = 'one;
            else
                next_state = 'zero;
        end
        'one:
        // we've seen one 1
        begin
            if #in then
                next_state = 'twos;
            else
                next_state = 'zero;
        end
        'twos:
        // we've seen at least 2 ones
        begin
            if #in then
                next_state = 'twos;
            else
                next_state = 'zero;
        end
endcase
```

Mealy Verilog FSM

```verilog
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 then
            out = 0;
        else
            out = #state;
endmodule
```

```verilog
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 then
            out = 0;
        else
            out = #state;
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
                begin
                    state = 'one; out = 1;
                    end
            end
        endcase
endmodule
```

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
- **Hardware description languages**