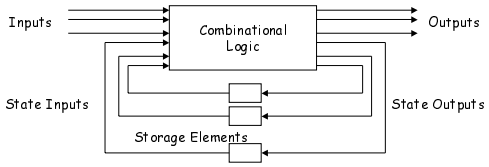
Sequential Logic Implementation

- 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

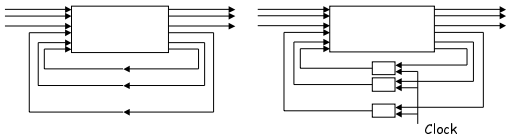
Abstraction of State Elements

- 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



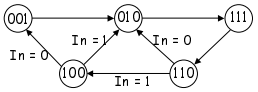
Forms of Sequential Logic

- 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

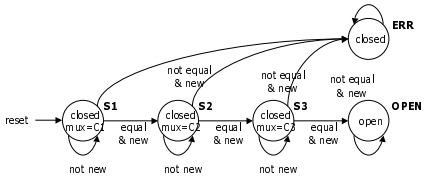
- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements



- Sequential Logic
  - Sequences through a series of states
  - Based on sequence of values on input signals
  - Clock period defines elements of sequence

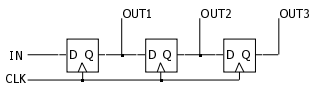
Example Finite State Machine Diagram

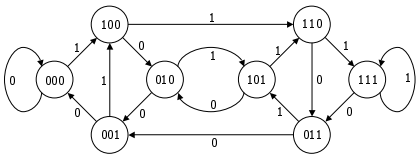
- Combination lock from first lecture



Can Any Sequential System be Represented with a State Diagram?

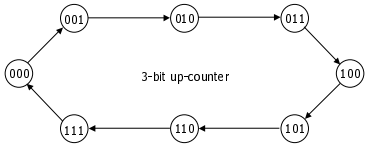
- Shift Register
  - Input value shown on transition arcs
  - Output values shown within state node





Counters are Simple Finite State Machines

- Counters
  - Proceed 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, …

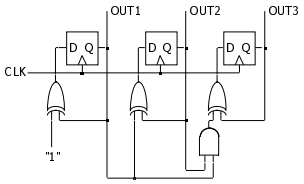


Verilog Upcounter

```
module binary_cntr (q, clk)
  inputs     clk;
  outputs    [2:0] q;
  reg        [2:0] q;
  reg        [2:0] p;

  always @(q)            //Calculate next state
    case (q)
      3'b000: p = 3'b001;
      3'b001: p = 3'b010;
      …
      3'b111: p = 3'b000;
    endcase

  always @(posedge clk)  //next becomes current state
    q <= p;

endmodule
```


How Do We 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

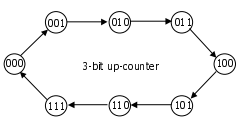


FSM Design Procedure

- 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

FSM Design Procedure: State Diagram to Encoded 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



current state		next state	
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	101	5
5	101	110	6
6	110	111	7
7	111	000	0

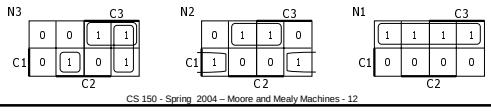
Implementation

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

C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

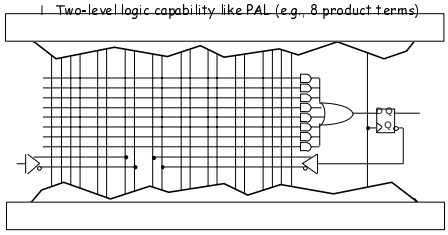
notation to show function represent input to D-FF

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



Implementation (cont'd)

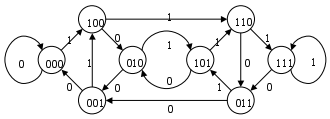
- Programmable Logic Building Block for Sequential Logic
  - Macro-cell: FF + logic
    - D-FF
    - Two-level logic capability like PAL (e.g., 8 product terms)



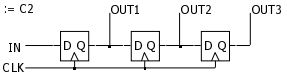
Another Example

- Shift Register
  - Input determines next state

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

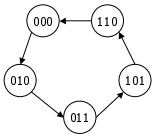


N1 := In
N2 := C1
N3 := C2



More Complex Counter Example

- Complex Counter
  - Repeats five 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

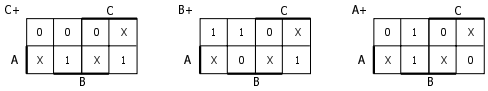


Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

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



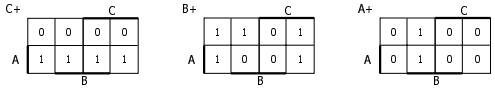
C+ := A

B+ := B' + A'C'

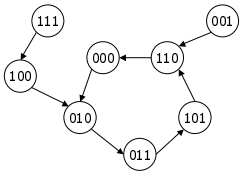
A+ := BC'

Self-Starting Counters (cont'd)

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

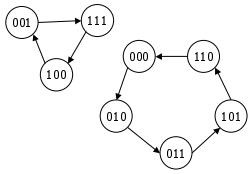


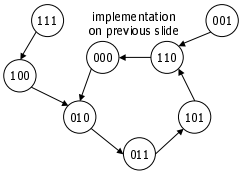
Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0



Self-Starting Counters

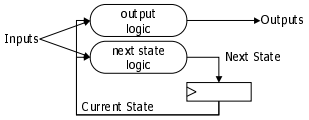
- Start-up States
  - At power-up, counter may be in an unused or invalid state
  - Designer must guarantee it (eventually) enters a valid state
- Self-starting Solution
  - Design counter so that invalid states eventually transition to a valid state
  - May limit exploitation of don't cares



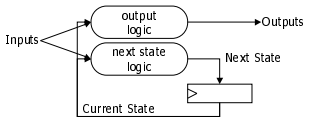


State Machine Model

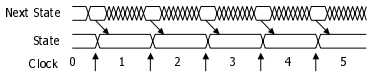
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - Next state
    - Function of current state and inputs
  - Outputs
    - Function of current state and inputs (Mealy machine)
    - Function of current state only (Moore machine)



State Machine Model (cont'd)

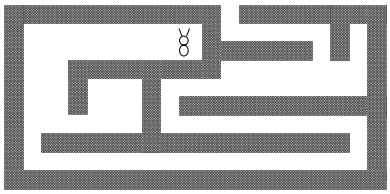


- States: S1, S2, ..., Sk
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: Fs(Si, Ij)
- Output function: Fo(Si) or Fo(Si, Ij)

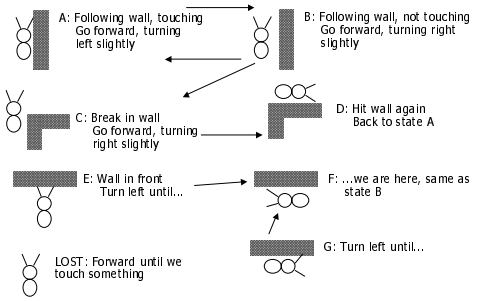


Example: Ant Brain (Ward, MIT)

- 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

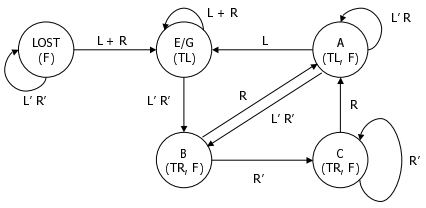


Ant Behavior



Designing an Ant Brain

- State Diagram

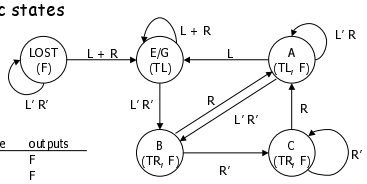


Synthesizing the Ant Brain Circuit

- 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



state	L	R	next state	outputs
LOST	0	0	LOST	F
LOST	–	1	E/G	F
LOST	1	–	E/G	F
A	0	0	B	TL, F
A	0	1	A	TL, F
A	1	–	E/G	TL, F
B	–	0	C	TR, F
B	–	1	A	TR, F
…	…	…	…	…

Synthesis

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

LOST - 000
E/G - 001
A - 010
B - 011
C - 100

state X,Y,Z	L	R	next state X', Y', Z'	outputs F	TR	TL
0 0 0	0	0	0 0 0	1	0	0
0 0 0	0	1	0 0 1	1	0	0
…	…	…	…	…		
0 1 0	0	0	0 1 1	1	0	1
0 1 0	0	1	0 1 0	1	0	1
0 1 0	1	0	0 0 1	1	0	1
0 1 0	1	1	0 0 1	1	0	1
0 1 1	0	0	1 0 0	1	1	0
0 1 1	0	1	0 1 0	1	1	0
…	…	…	…	…		

it now remains to synthesize these 6 functions

Synthesis of Next State and Output Functions

state X,Y,Z	inputs L	R	next state X+,Y+,Z+	outputs F	TR	TL
0 0 0	0	0	0 0 0	1	0	0
0 0 0	-	1	0 0 1	1	0	0
0 0 0	1	-	0 0 1	1	0	0
0 0 1	0	0	0 1 1	0	0	1
0 0 1	-	1	0 1 0	0	0	1
0 0 1	1	-	0 1 0	0	0	1
0 1 0	0	0	0 1 1	1	0	1
0 1 0	0	1	0 1 0	1	0	1
0 1 0	1	-	0 0 1	1	0	1
0 1 1	-	0	1 0 0	1	1	0
0 1 1	-	1	0 1 0	1	1	0
1 0 0	-	0	1 0 0	1	1	0
1 0 0	-	1	0 1 0	1	1	0

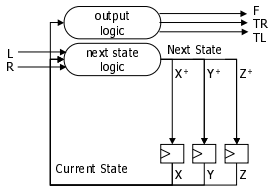
e.g.

$TR = X + Y Z$

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

Circuit Implementation

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

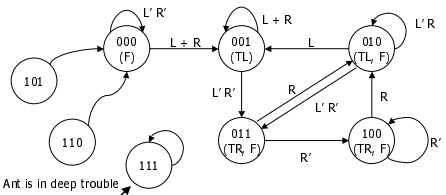


Verilog Sketch

```
module ant_brain (F, TR, TL, L, R)
  inputs     L, R;
  outputs    F, TR, TL;
  reg        X, Y, Z;

  assign F  = function(X, Y, Z, L, R);
  assign TR = function(X, Y, Z, L, R);
  assign TL = function(X, Y, Z, L, R);

  always @(posedge clk)
    begin
      X <= function (X, Y, Z, L, R);
      Y <= function (X, Y, Z, L, R);
      Z <= function (X, Y, Z, L, R);
    end
  endmodule
```


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



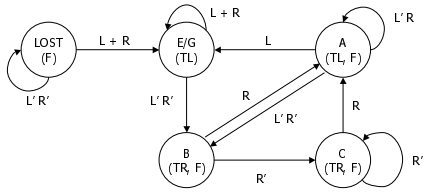
Ant is in deep trouble if it gets in this state

State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two state 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

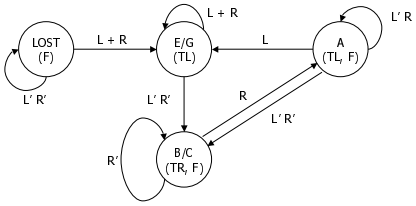
Ant Brain Revisited

- Any equivalent states?



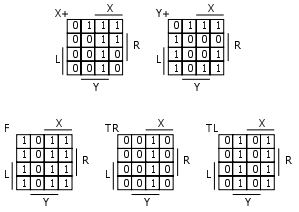
New Improved Brain

- 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

state X,Y	inputs L R	next state X',Y'	outputs F	TR	TL
0 0	0 0	0 0	1	0	0
0 0	- 1	0 1	1	0	0
0 0	1 -	0 1	1	0	0
0 1	0 0	1 1	0	0	1
0 1	- 1	0 1	0	0	1
0 1	1 -	0 1	0	0	1
1 0	0 0	1 1	1	0	1
1 0	0 1	1 0	1	0	1
1 0	1 -	0 1	1	0	1
1 1	- 0	1 1	1	1	0
1 1	- 1	1 0	1	1	0



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