EECS 150 - Components and Design Techniques for Digital Systems

FSMs
9/11/2007

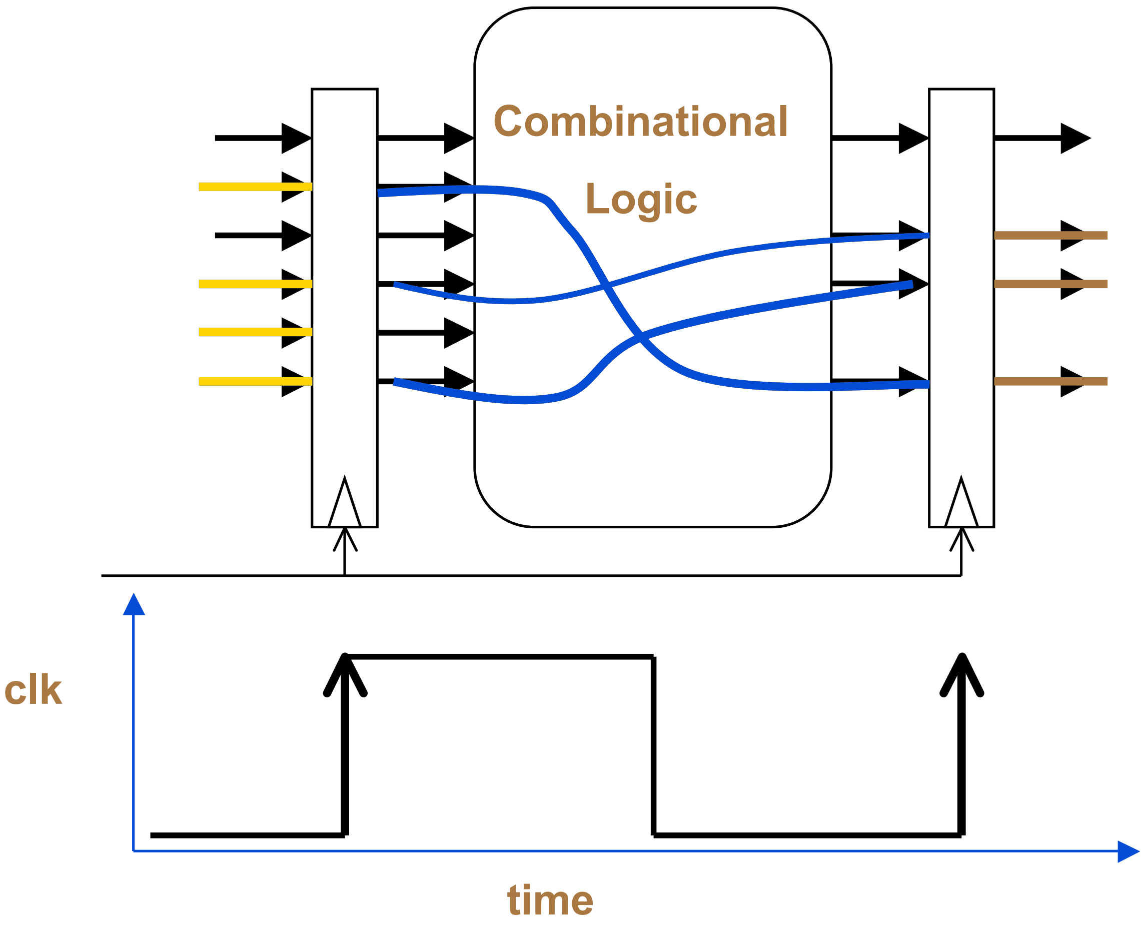
Sarah Bird
Electrical Engineering and Computer Sciences
University of California, Berkeley

Slides borrowed from David Culler Fa04 and Randy Katz Sp07

Sequential Logic Implementation

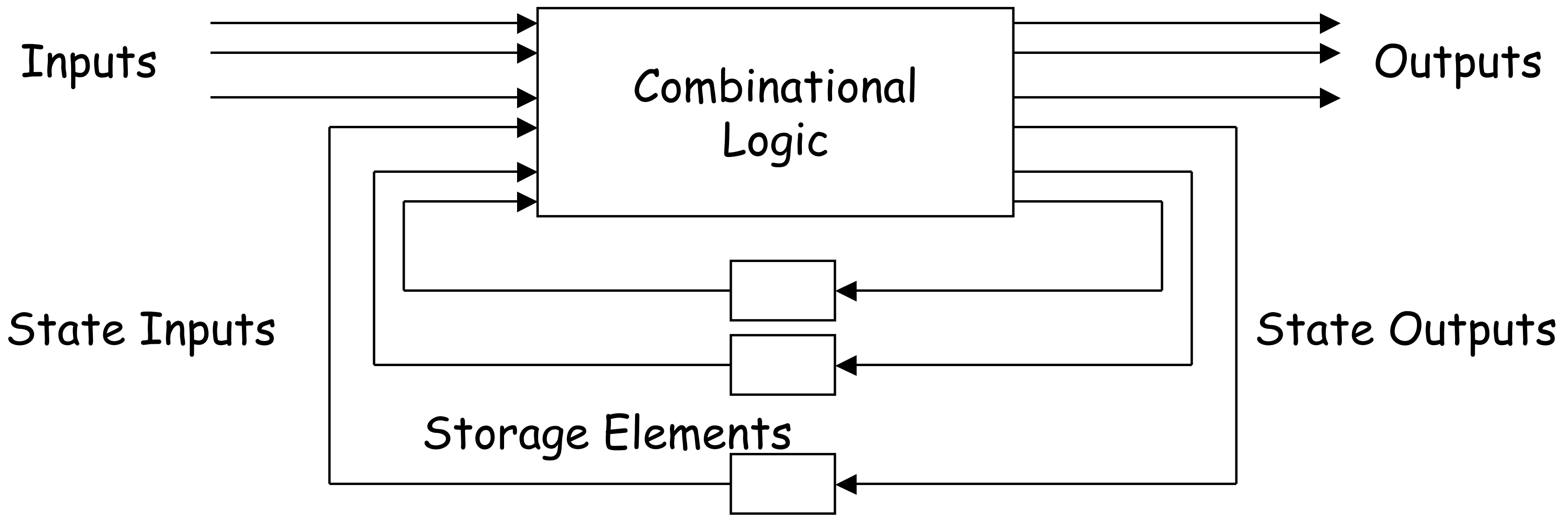
- Models for representing sequential circuits
  - Finite-state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)
- Design procedure
  - State diagrams
  - State transition table
  - Next state functions

Recall: What makes Digital Systems tick?



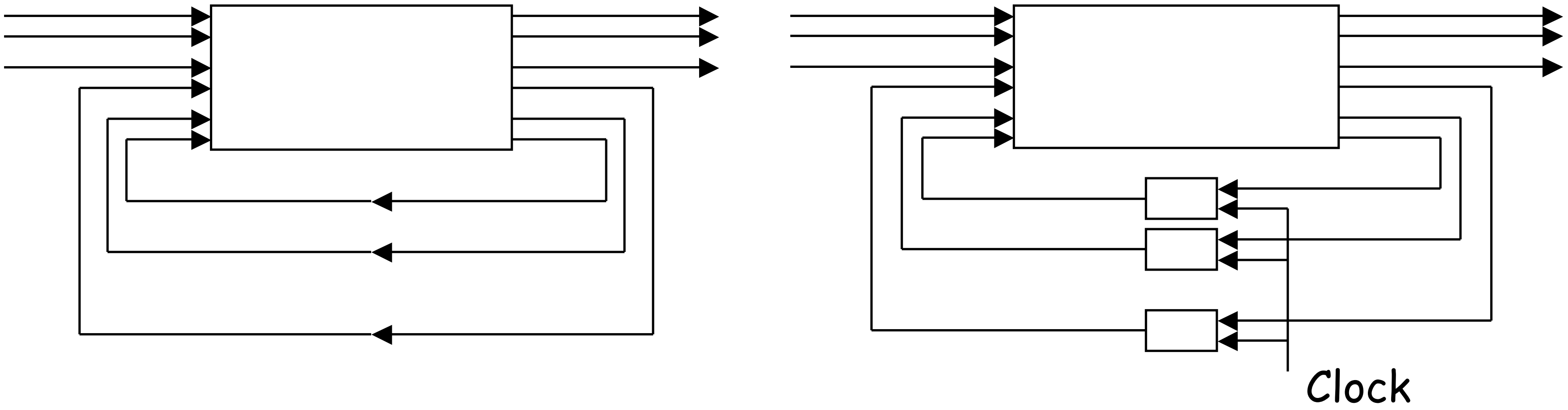
Abstraction of State Elements

- 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



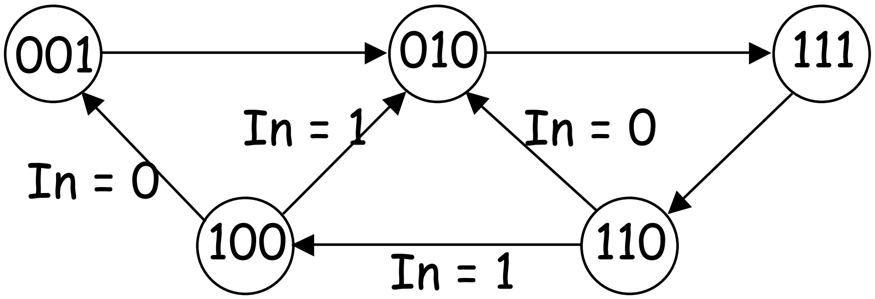
Forms of Sequential Logic

- 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

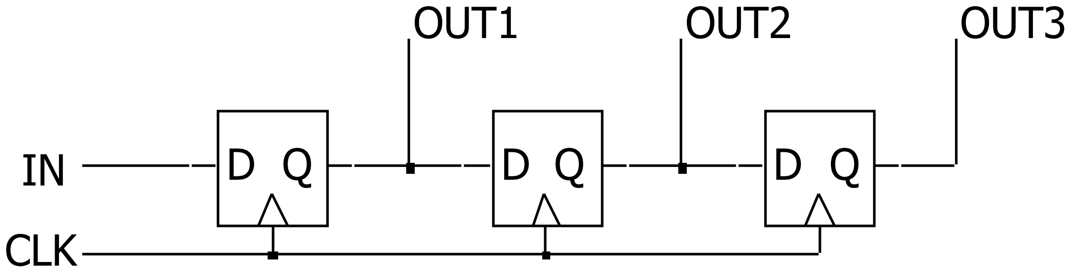
- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements

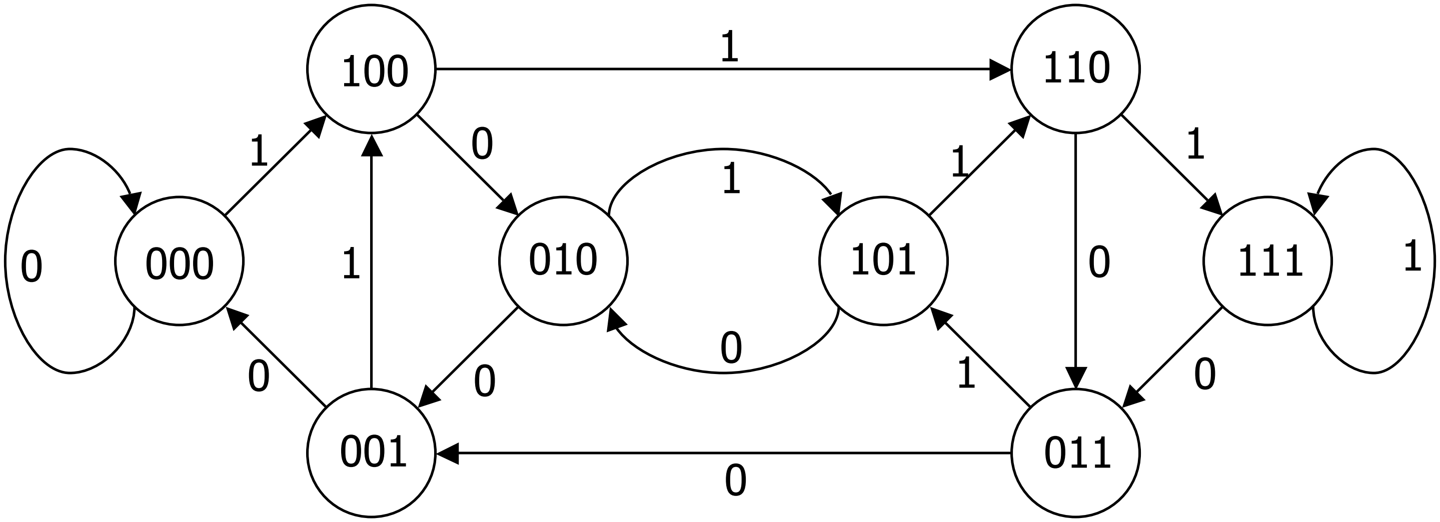


- Sequential Logic
  - Sequences through a series of states
  - Based on sequence of values on input signals
  - Clock period defines elements of sequence

Can Any Sequential System be Represented with a State Diagram?

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

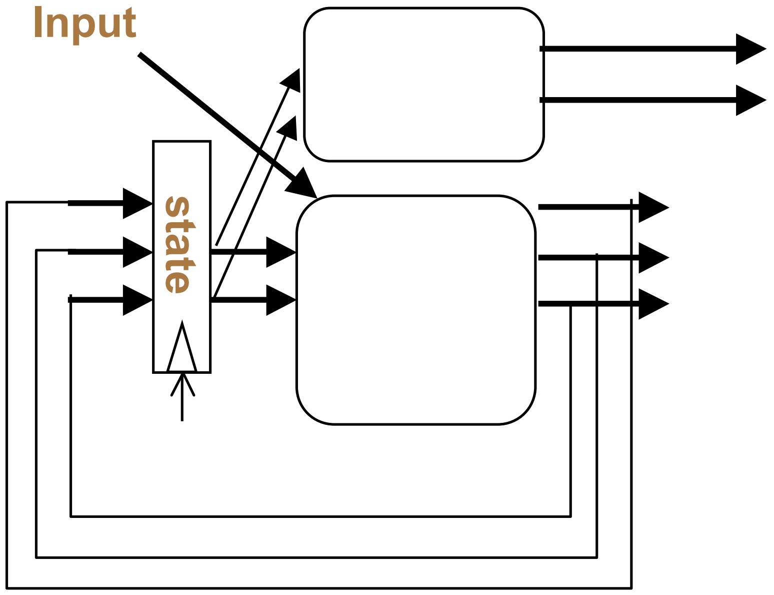




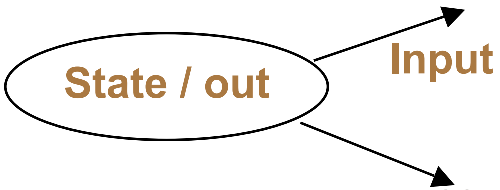
Two Kinds of FSMs

- Moore Machine vs Mealy Machine

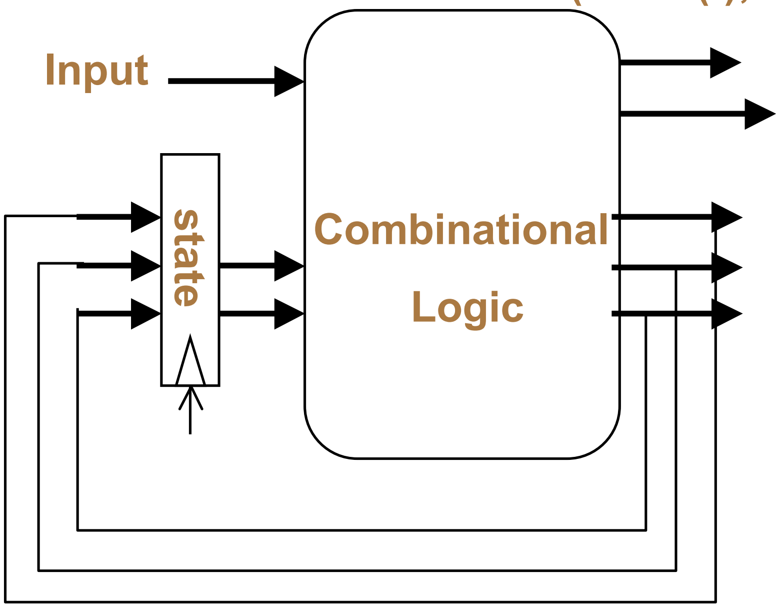
Output (t) = *G*(state(t))



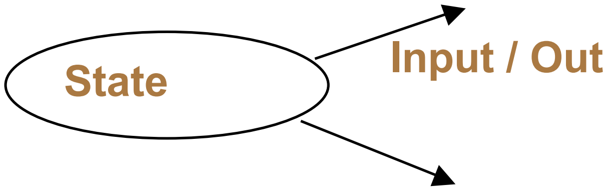
state(t+1) = *F* (state(t), input(t))



Output (t) = *G*(state(t), Input)

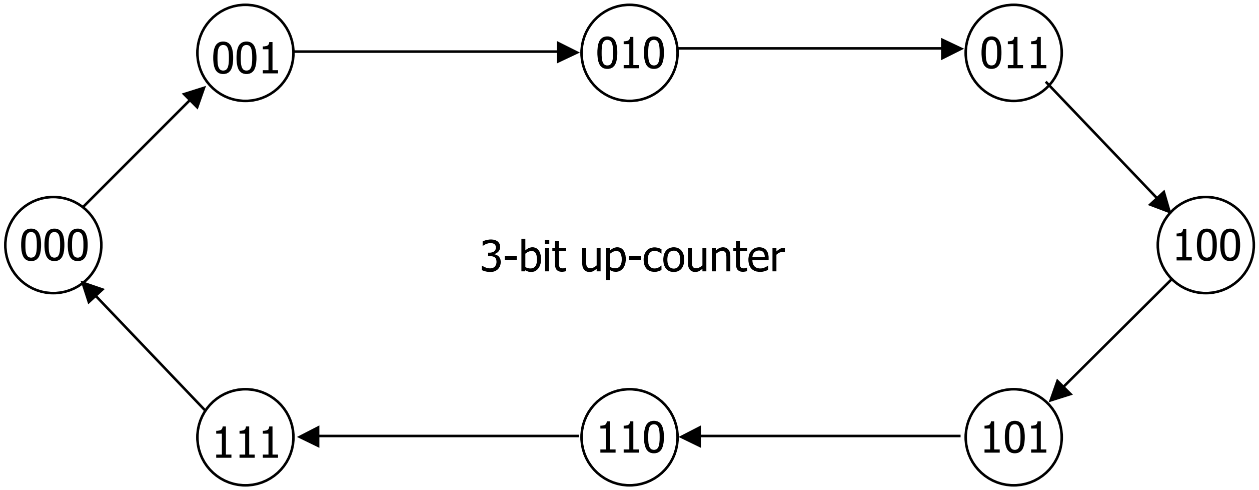


state(t+1) = *F* (state(t), input)



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, ...

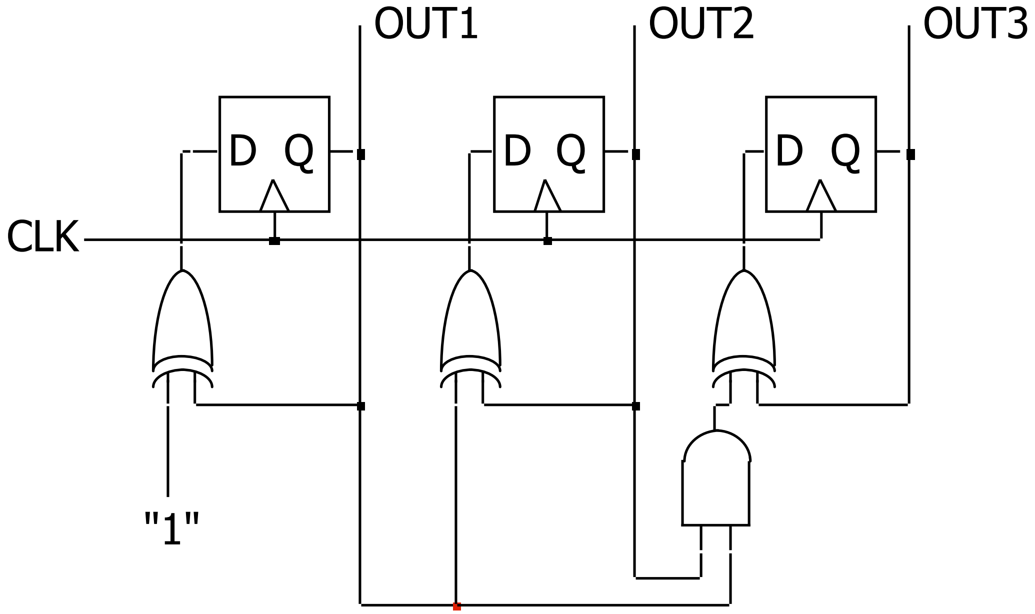


Verilog Upcounter

```
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

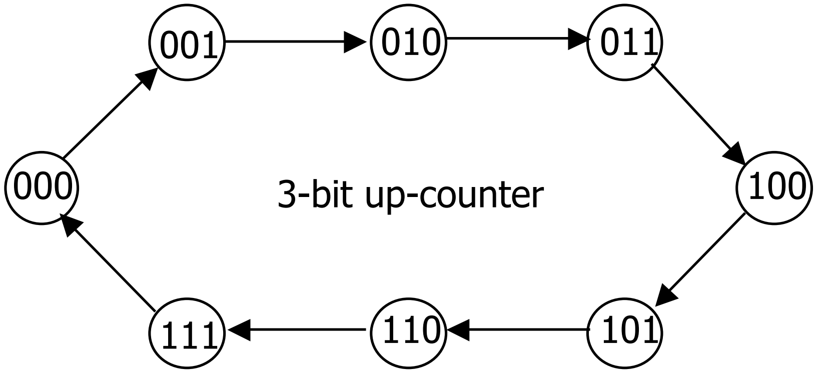


FSM Design Procedure

- 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

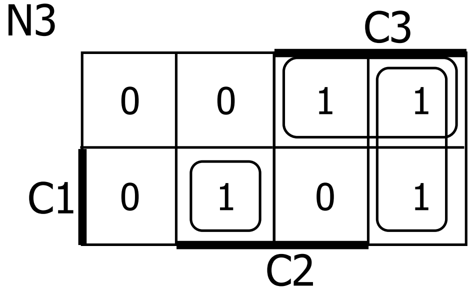
Implementation

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

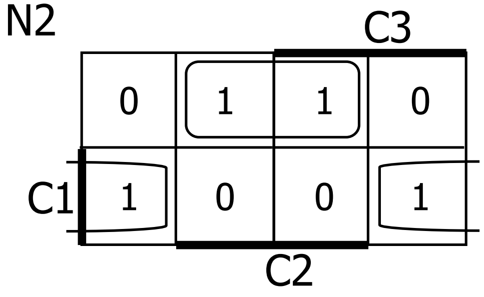
C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

notation to show function represent input to D-FF

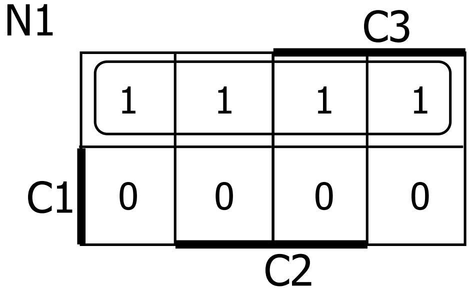
N1 := C1'
N2 := C1C2' + C1'C2
:= C1 xor C2
N3 := C1C2C3' + C1'C3 + C2'C3
:= C1C2C3' + (C1' + C2')C3
:= (C1C2) xor C3



0	0	1	1
0	1	0	1

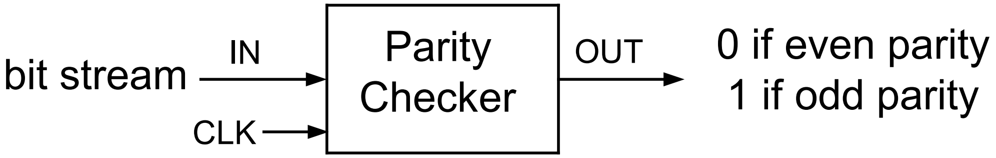


0	1	1	0
1	0	0	1



1	1	1	1
0	0	0	0

Parity Checker FSM



bit stream

IN

CLK

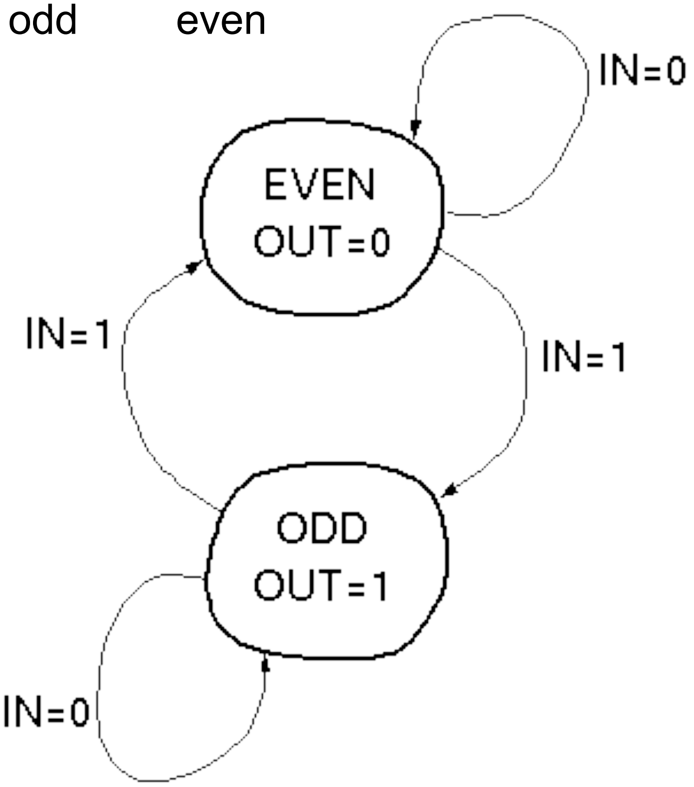
Parity Checker

OUT

0 if even parity
1 if odd parity

example:	0	0	1	1	1	0	1
	even	even	odd	even	odd	odd	even

time

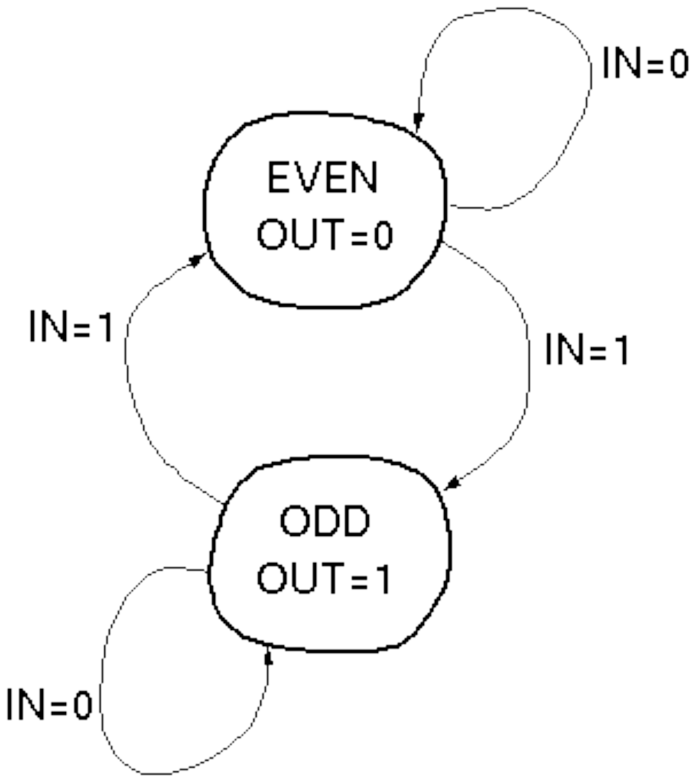


- "State Transition Diagram"
  - circuit is in one of two states.
  - transition on each cycle with each new input, over exactly one arc (edge).
  - Output depends on which state the circuit is in.

Formal Design Process

- State Transition Table:

present state	OUT	IN	next state
EVEN	0	0	EVEN
EVEN	0	1	ODD
ODD	1	0	ODD
ODD	1	1	EVEN



- Invent a code to represent states:

  Let 0 = EVEN state, 1 = ODD state

present state (ps)	OUT	IN	next state (ns)
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	0

Derive logic equations from table (how?):

OUT = PS

NS = PS xor IN

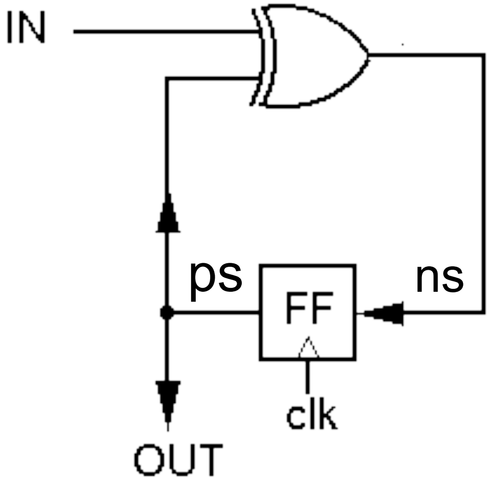
Formal Design Process

Logic equations from table:

OUT = PS

NS = PS xor IN

- Circuit Diagram:



  - XOR gate for ns calculation
  - DFF to hold present state
  - no logic needed for output

- Review of Design Steps:

1. Circuit functional specification
2. State Transition Diagram
3. Symbolic State Transition Table
4. Encoded State Transition Table
5. Derive Logic Equations
6. Circuit Diagram

   FFs for state

   CL for NS and OUT

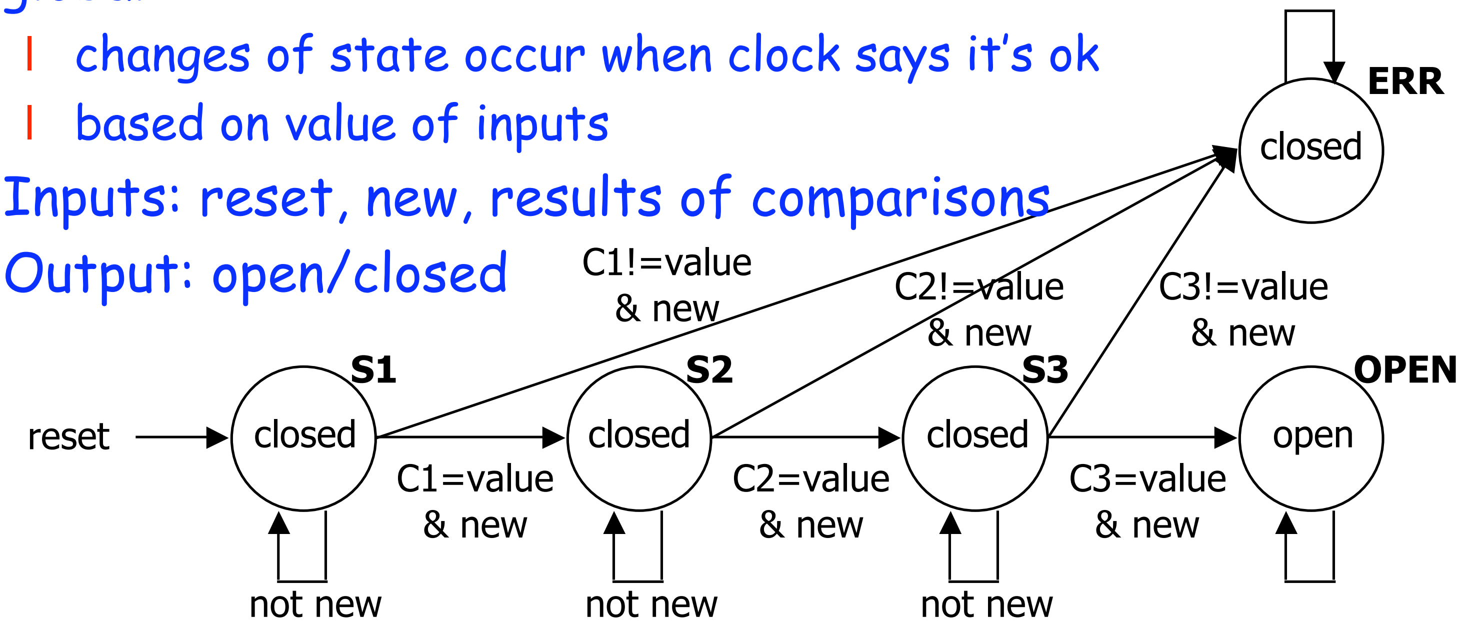
Another example

- Door combination lock:
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset

  - inputs: sequence of input values, reset
  - outputs: door open/close
  - memory: must remember combination
    or always have it available as an input

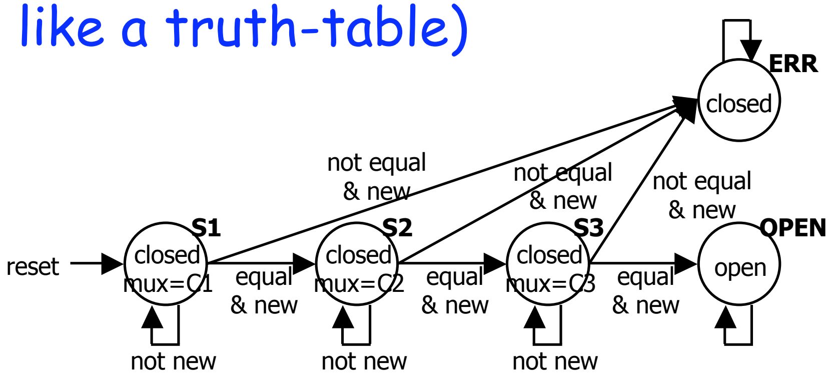
Sequential example: abstract control

- Finite-state diagram
  - States: 5 states
    - represent point in execution of machine
    - each state has outputs
  - Transitions: 6 from state to state, 5 self transitions, 1 global
    - changes of state occur when clock says it's ok
    - based on value of inputs
  - Inputs: reset, new, results of comparisons
  - Output: open/closed



Sequential example (cont'd): finite-state machine

- Finite-state machine
  - generate state table (much like a truth-table)



Symbolic states

reset	new	equal	state	next state	mux	open/closed
1	–	–	–	S1	C1	closed
0	0	–	S1	S1	C1	closed
0	1	0	S1	ERR	–	closed
0	1	1	S1	S2	C2	closed
0	0	–	S2	S2	C2	closed
0	1	0	S2	ERR	–	closed
0	1	1	S2	S3	C3	closed
0	0	–	S3	S3	C3	closed
0	1	0	S3	ERR	–	closed
0	1	1	S3	OPEN	–	closed
0	–	–	OPEN	OPEN	–	open
0	–	–	ERR	ERR	–	closed

Encoding?

Sequential example: encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - and as many as 5: 00001, 00010, 00100, 01000, 10000
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000

binary

One-hot

hybrid

- Encode outputs
  - output mux can be: C1, C2, or C3
    - needs 2 to 3 bits to encode
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - needs 1 or 2 bits to encode
    - choose 1 bits: 1, 0

Sequential example (cont'd): encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - choose 1 bits: 1, 0

				next		
reset	new	equal	state	state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
0	0	–	0010	0010	010	0
0	1	0	0010	0000	–	0
0	1	1	0010	0100	100	0
0	0	–	0100	0100	100	0
0	1	0	0100	0000	–	0
0	1	1	0100	1000	–	1
0	–	–	1000	1000	–	1
0	–	–	0000	0000	–	0

good choice of encoding!

mux is identical to last 3 bits of next state

open/closed is identical to first bit of 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

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