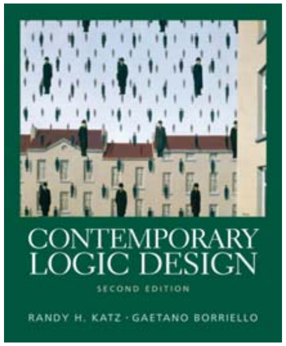
CSE 370 Spring 2006
Introduction to Digital Design

Lecture 17: Introduction to Finite State Machines

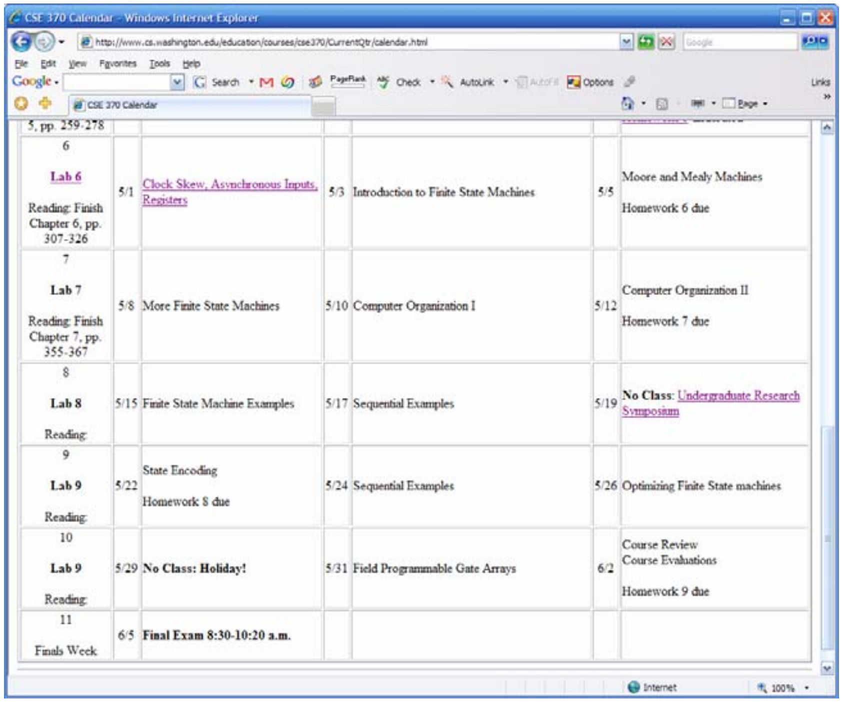


Last Lecture

- Clock Skew
- Asynchronous Inputs
- Registers

Today

- Finite State Machines


CSE 370 Calendar - Windows Internet Explorer
http://www.cs.washington.edu/education/courses/cse370/CurrentQtr/calendar.html
File Edit View Favorites Tools Help
Google
Search
PageRank
Check
AutoLink
AutoFill
Options
Links
CSE 370 Calendar
Page
Internet
100%

5, pp. 259-278						
6 Lab 6 Reading: Finish Chapter 6, pp. 307-326	5/1	Clock Skew, Asynchronous Inputs, Registers	5/3	Introduction to Finite State Machines	5/5	Moore and Mealy Machines Homework 6 due
7 **Lab 7** Reading: Finish Chapter 7, pp. 355-367	5/8	More Finite State Machines	5/10	Computer Organization I	5/12	Computer Organization II Homework 7 due
8 **Lab 8** Reading:	5/15	Finite State Machine Examples	5/17	Sequential Examples	5/19	**No Class**: Undergraduate Research Symposium
9 **Lab 9** Reading:	5/22	State Encoding Homework 8 due	5/24	Sequential Examples	5/26	Optimizing Finite State machines
10 **Lab 9** Reading:	5/29	**No Class: Holiday!**	5/31	Field Programmable Gate Arrays	6/2	Course Review Course Evaluations Homework 9 due
11 Finals Week	6/5	**Final Exam 8:30-10:20 a.m.**				

Administrivia

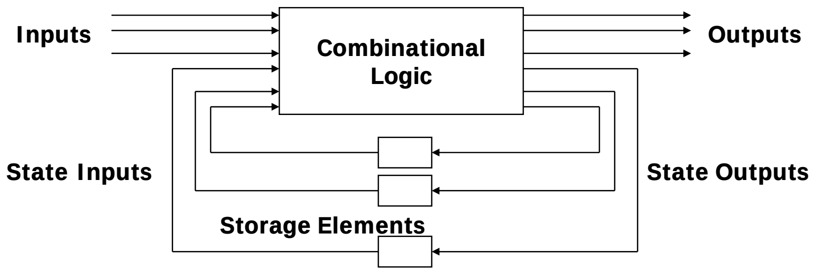
- Homework 6 due Friday

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

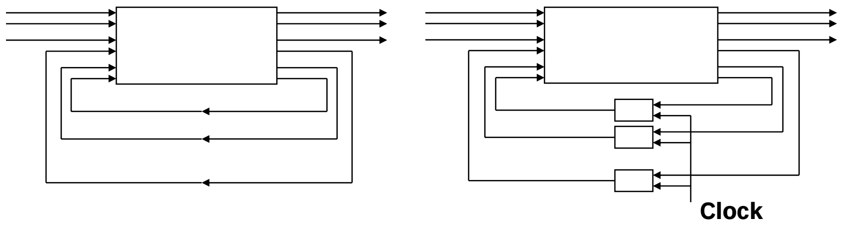
Abstraction of state elements

- 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



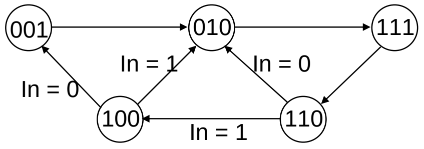
Forms of sequential logic

- 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)



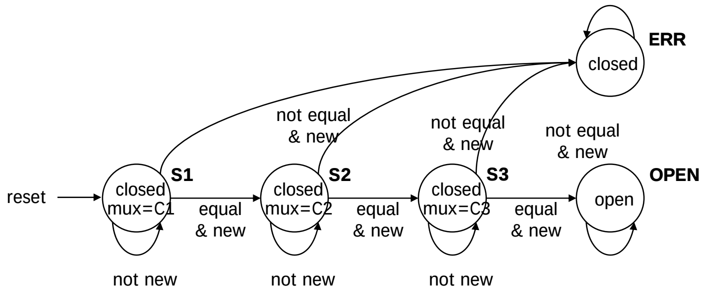
Finite state machine representations

- 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



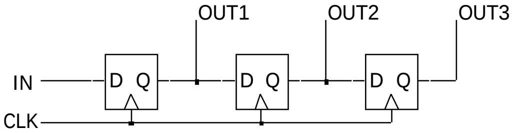
Example finite state machine diagram

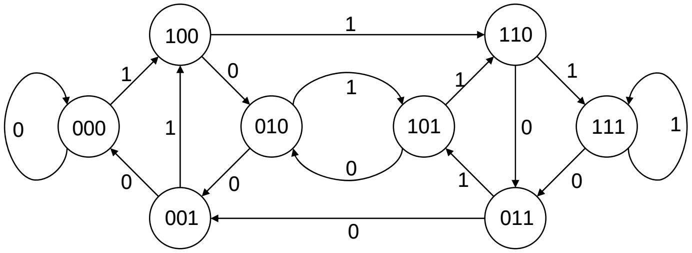
- Combination lock from earlier
  - 5 states
  - 5 self-transitions
  - 6 other transitions between states
  - 1 reset transition (from all states) to state S1



Can any sequential system be represented with a state diagram?

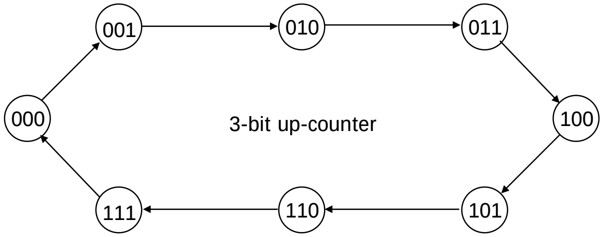
- Shift register
  - input value shown on transition arcs
  - output values shown within state node





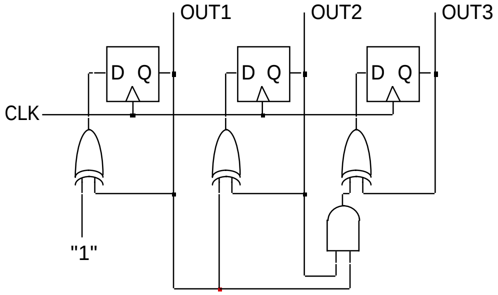
Counters are simple finite state machines

- 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

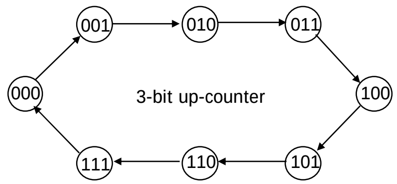


FSM design procedure

- 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

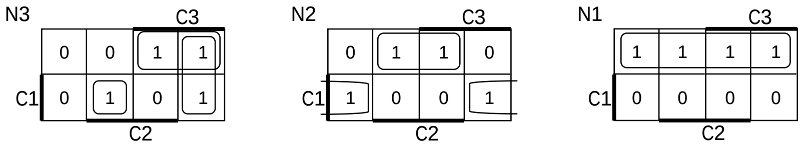
Implementation

- 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

Verilog notation to show function represents an input to D-FF

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



N3

0	0	1	1
0	1	0	1

N2

0	1	1	0
1	0	0	1

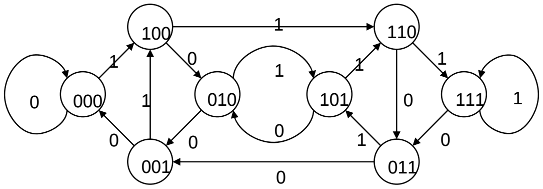
N1

1	1	1	1
0	0	0	0

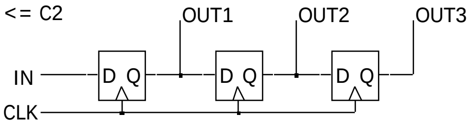
Back to the shift register

- 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

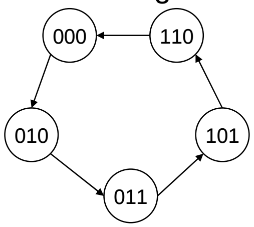


N1 <= In
N2 <= C1
N3 <= C2



More complex counter example

- Complex counter
  - repeats 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

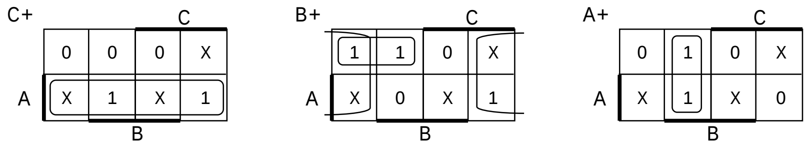


Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

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+

	0	0	0	X
A	X	1	X	1

B+

	1	1	0	X
A	X	0	X	1

A+

	0	1	0	X
A	X	1	X	0

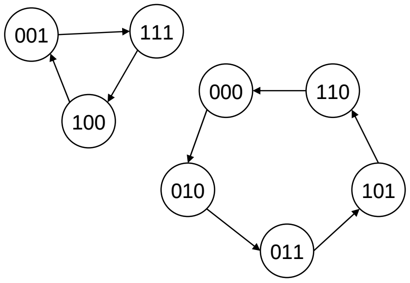
C+ <= A

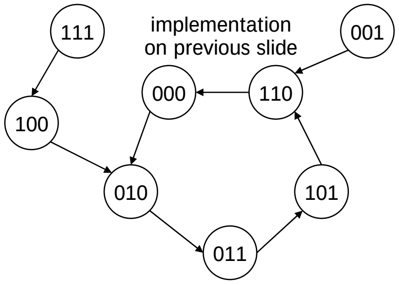
B+ <= B' + A'C'

A+ <= BC'

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

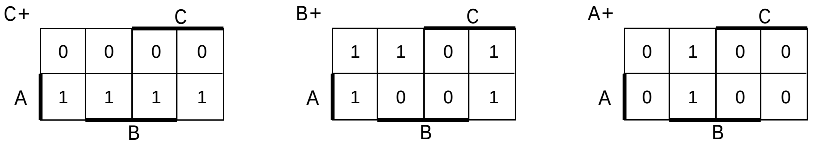




implementation on previous slide

Self-starting counters (cont'd)

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



C+

	0	0	0	0
A	1	1	1	1

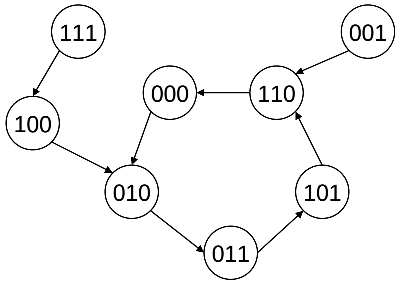
B+

	1	1	0	1
A	1	0	0	1

A+

	0	1	0	0
A	0	1	0	0

Present State			Next State		
C	B	A	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



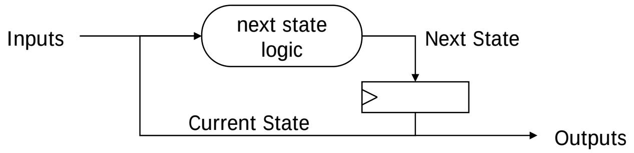
Activity

- 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)

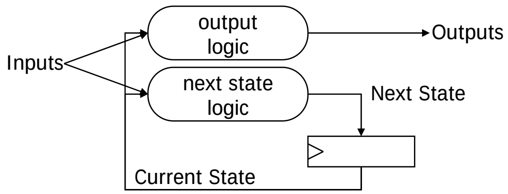
Counter/shift-register model

- 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)


output logic
Outputs
Inputs
next state logic
Next State
Current State