Lecture 15

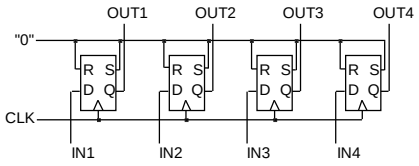
- Registers
- Counters
- Finite State Machine (FSM) design

Registers

- Group of storage elements read/written as a unit
  - Store related values (e.g. a binary word)
- Collection of flip-flops with common control
  - Share clock, reset, set lines
- Example:
  - Storage registers
  - Shift registers
  - Counters

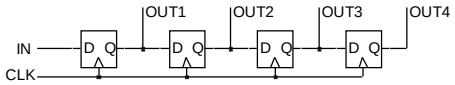
Storage registers

- Basic storage registers use flip-flops
- Example: 4 bit storage register



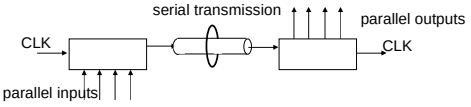
Shift registers

- Hold successively sampled input values
  - Delays values in time
  - Example: 4-bit shift register
    - Stores 4 input values in sequence

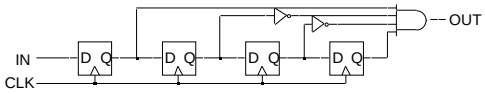


Shift register applications

- Parallel-to-serial conversion for signal transmission

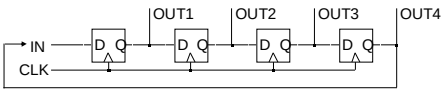


- Pattern recognition (circuit recognizes 1001)



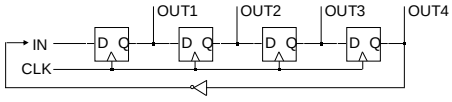
Counters: Ring counter

- Ring counter: Sequence is 1000, 0100, 0010, 0001
  - Assuming one of these patterns is the starting state



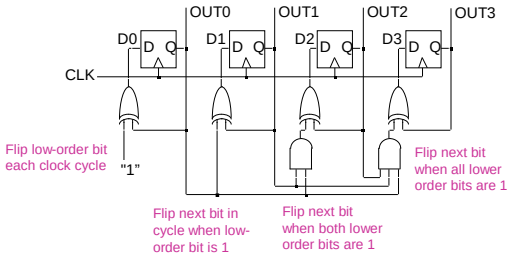
Counters: Johnson counter

- Johnson counter: Sequence is 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000



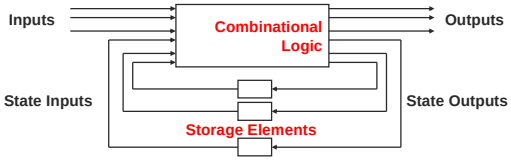
Counters: Binary counter

- Has logic between flip-flops



Storing "states" for FSMs

- Combinational logic and storage elements
  - Localized feedback loops
  - Choice of storage elements alters the logic



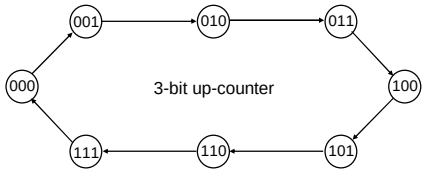
Finite-state machines (FSMs)

- States: Possible storage-element values
- Transitions: Changes in state
  - Clock synchronizes the state changes
- Sequential logic
  - Sequences through a series of states
  - Based on inputs and present state

FSM design example: Counters

1. Draw a state diagram
2. Draw a state-transition table
3. Encode the next-state functions
   - Minimize the logic using k-maps
4. Implement the design

1. Draw a state diagram



2. Draw a state-transition table

- Like a truth-table
  - State encoding is easy for counters → Use count 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

3. Encode next state functions

- Assume D flip-flops as state elements

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

N1

C1 \ C3	1	1	1	1
0	0	0	0	

C2

N2

C1 \ C3	0	1	1	0
1	0	0	1	

C2

N3

C1 \ C3	0	0	1	1
0	1	0	1	

C2

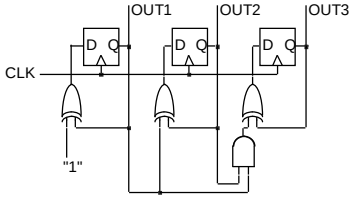
N1 := C1'

N2 := C1C2' + C1'C2
:= C1 xor C2

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

4. Implement the design

- 3 flip-flops hold state
  - Counter is synchronously clocked
- Minimized logic computes next state



What if we use T flip-flops?

T flip-flops

Ti = 1 iff Ni ≠ Ci

T0 := 1
T1 := C0
T2 := C0 C1

C2	C1	C0	N2	N1	N0	T2	T1	T0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

T0

C0 \ C2	1	1	1	1
1	1	1	1	

C1

T1

C0 \ C2	0	0	0	0
1	1	1	1	

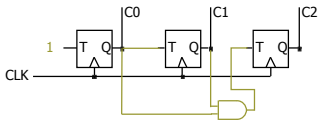
C1

T2

C0 \ C2	0	0	0	0
0	1	1	0	

C1

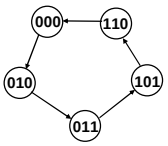
4. Implement the design



Example: 5-state counter

- Counter repeats 5 states in sequence
  - Sequence is 000, 010, 011, 101, 110, 000

Step 1: State diagram



Step 2: State transition table
Assume D flip-flops

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

3. Encode next state functions

C+

A \ BC	00	01	11	10
0	0	0	X	0
1	X	1	X	1

C+ = A

B+

A \ BC	00	01	11	10
0	1	1	X	0
1	X	0	X	1

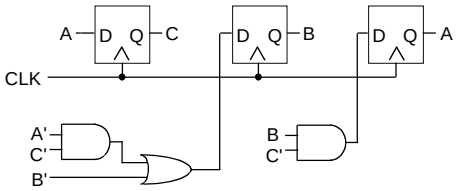
B+ = B' + A'C'

A+

A \ BC	00	01	11	10
0	0	1	X	0
1	X	1	X	0

A+ = BC'

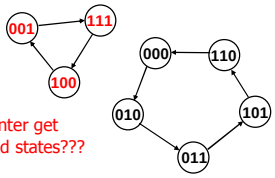
4. Implement the design



Recall that a D flip flop also produces Q' so A', B', and C' would all be available without any extra inverters

Is our design robust?

- What if the counter starts in a 111 state?



Does our counter get stuck in invalid states???

5-state counter

- Back-annotate our design to check it

Fill in state transition table

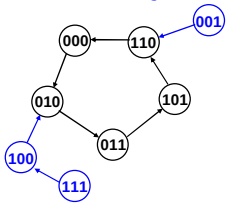
Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0

A+ = BC'

B+ = B' + A'C'

C+ = A

Draw state diagram



The proper methodology is to ***design*** your counter to be self-starting

Self-starting counters

- Invalid states should always transition to valid states
  - Assures startup
  - Assures bit-error tolerance
- Design your counters to be self-starting
  - Draw all states in the state diagram
  - Fill in the entire state-transition table
  - May limit your ability to exploit don't cares
    - Choose startup transitions that minimize the logic