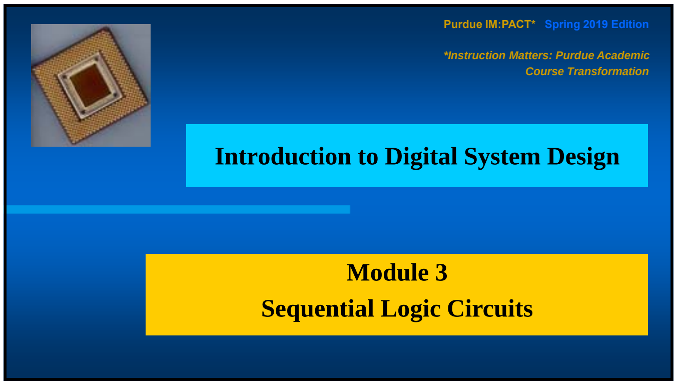
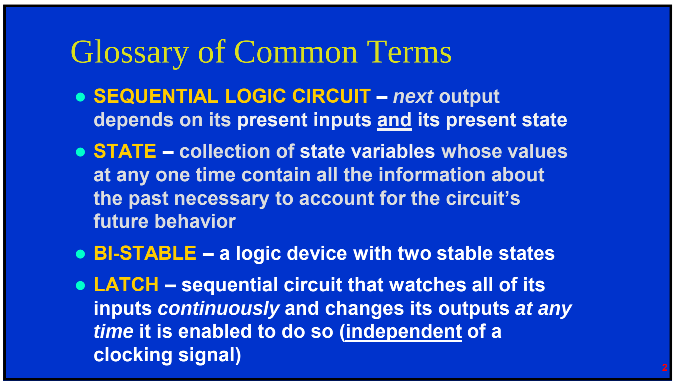


Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

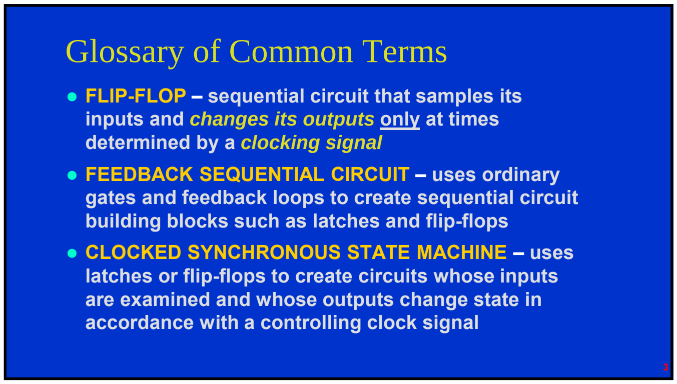
Introduction to Digital System Design

Module 3
Sequential Logic Circuits



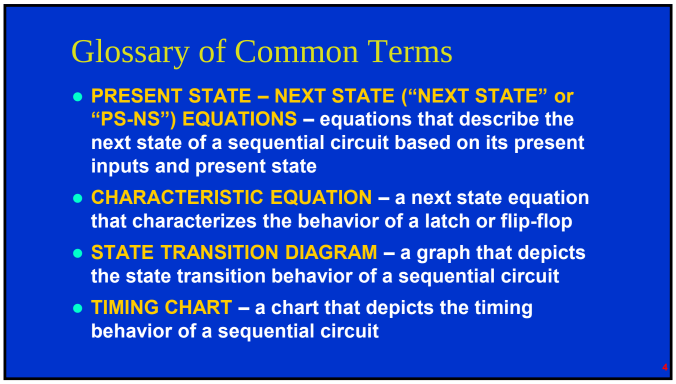
Glossary of Common Terms

- **SEQUENTIAL LOGIC CIRCUIT** – *next* output depends on its present inputs and its present state
- **STATE** – collection of state variables whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior
- **BI-STABLE** – a logic device with two stable states
- **LATCH** – sequential circuit that watches all of its inputs *continuously* and changes its outputs *at any time* it is enabled to do so (independent of a clocking signal)



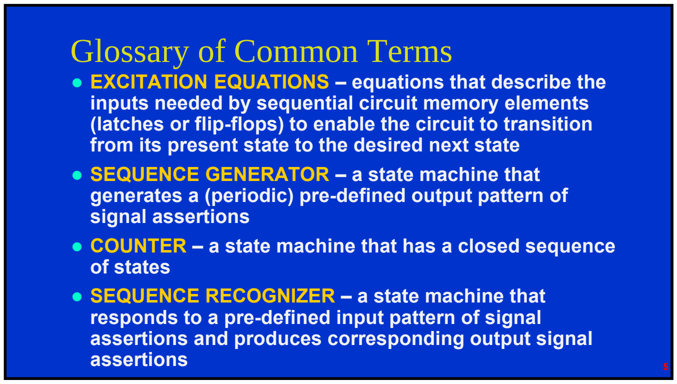
Glossary of Common Terms

- **FLIP-FLOP** – sequential circuit that samples its inputs and *changes its outputs* only at times determined by a *clocking signal*
- **FEEDBACK SEQUENTIAL CIRCUIT** – uses ordinary gates and feedback loops to create sequential circuit building blocks such as latches and flip-flops
- **CLOCKED SYNCHRONOUS STATE MACHINE** – uses latches or flip-flops to create circuits whose inputs are examined and whose outputs change state in accordance with a controlling clock signal



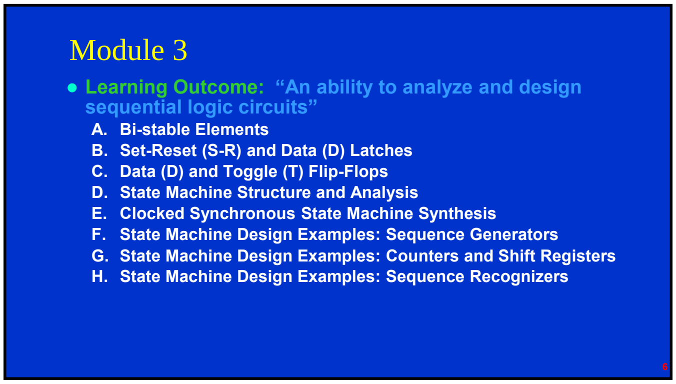
Glossary of Common Terms

- **PRESENT STATE – NEXT STATE ("NEXT STATE" or "PS-NS") EQUATIONS** – equations that describe the next state of a sequential circuit based on its present inputs and present state
- **CHARACTERISTIC EQUATION** – a next state equation that characterizes the behavior of a latch or flip-flop
- **STATE TRANSITION DIAGRAM** – a graph that depicts the state transition behavior of a sequential circuit
- **TIMING CHART** – a chart that depicts the timing behavior of a sequential circuit



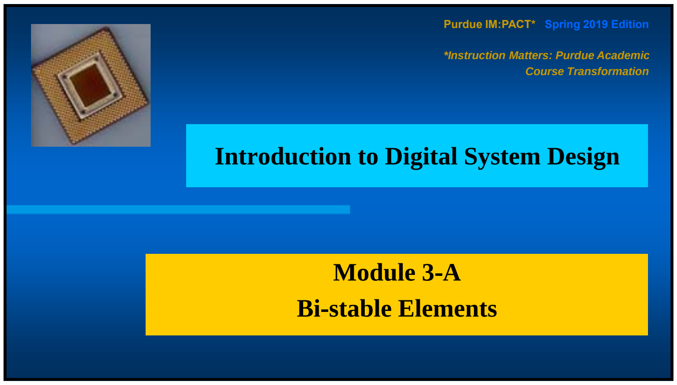
Glossary of Common Terms

- **EXCITATION EQUATIONS** – equations that describe the inputs needed by sequential circuit memory elements (latches or flip-flops) to enable the circuit to transition from its present state to the desired next state
- **SEQUENCE GENERATOR** – a state machine that generates a (periodic) pre-defined output pattern of signal assertions
- **COUNTER** – a state machine that has a closed sequence of states
- **SEQUENCE RECOGNIZER** – a state machine that responds to a pre-defined input pattern of signal assertions and produces corresponding output signal assertions



Module 3

- Learning Outcome: "An ability to analyze and design sequential logic circuits"
  - A. Bi-stable Elements
  - B. Set-Reset (S-R) and Data (D) Latches
  - C. Data (D) and Toggle (T) Flip-Flops
  - D. State Machine Structure and Analysis
  - E. Clocked Synchronous State Machine Synthesis
  - F. State Machine Design Examples: Sequence Generators
  - G. State Machine Design Examples: Counters and Shift Registers
  - H. State Machine Design Examples: Sequence Recognizers

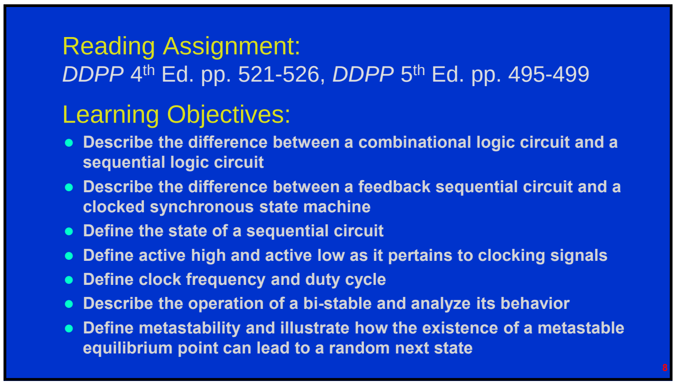


Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 3-A
Bi-stable Elements



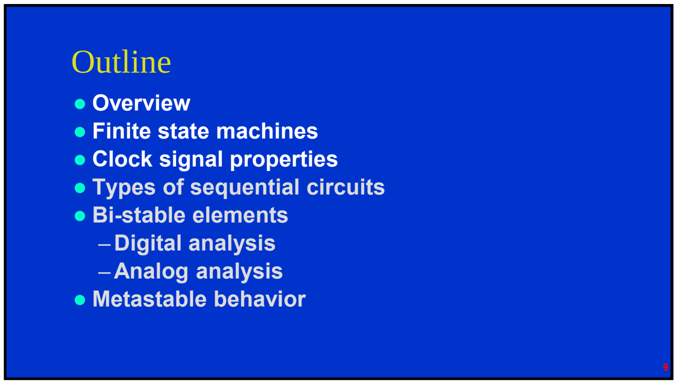
Reading Assignment:

DDPP 4th Ed. pp. 521-526, *DDPP* 5th Ed. pp. 495-499

Learning Objectives:

- Describe the difference between a combinational logic circuit and a sequential logic circuit
- Describe the difference between a feedback sequential circuit and a clocked synchronous state machine
- Define the state of a sequential circuit
- Define active high and active low as it pertains to clocking signals
- Define clock frequency and duty cycle
- Describe the operation of a bi-stable and analyze its behavior
- Define metastability and illustrate how the existence of a metastable equilibrium point can lead to a random next state

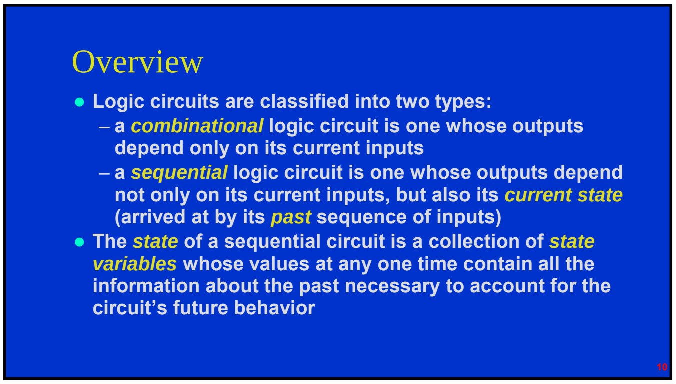
8



Outline

- Overview
- Finite state machines
- Clock signal properties
- Types of sequential circuits
- Bi-stable elements
  - Digital analysis
  - Analog analysis
- Metastable behavior

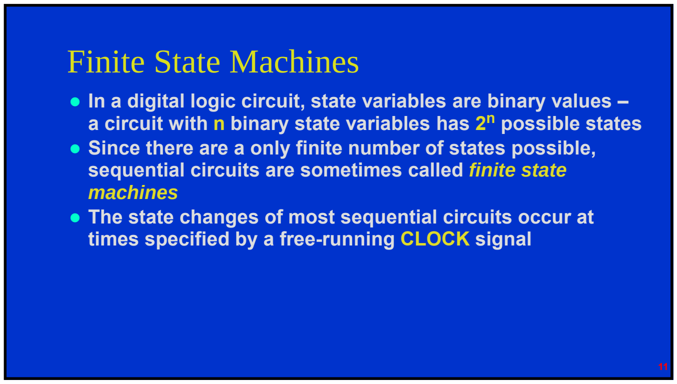
9



Overview

- Logic circuits are classified into two types:
  - a *combinational* logic circuit is one whose outputs depend only on its current inputs
  - a *sequential* logic circuit is one whose outputs depend not only on its current inputs, but also its *current state* (arrived at by its *past* sequence of inputs)
- The *state* of a sequential circuit is a collection of *state variables* whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior

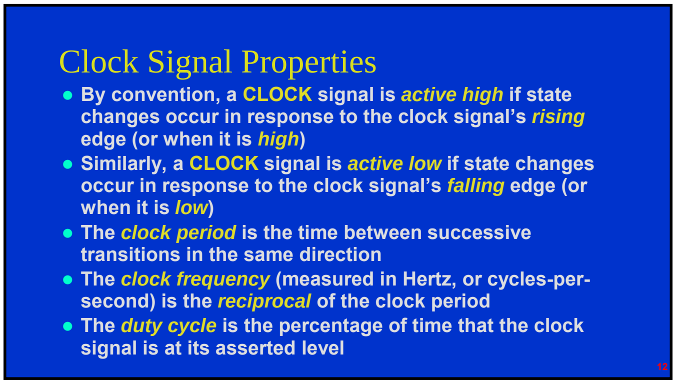
10



Finite State Machines

- In a digital logic circuit, state variables are binary values – a circuit with n binary state variables has 2^n possible states
- Since there are a only finite number of states possible, sequential circuits are sometimes called *finite state machines*
- The state changes of most sequential circuits occur at times specified by a free-running CLOCK signal

11

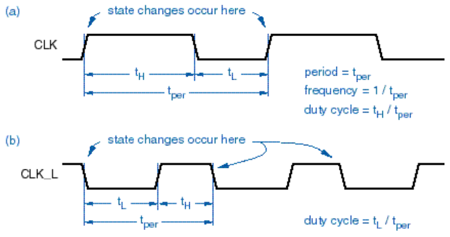


Clock Signal Properties

- By convention, a CLOCK signal is *active high* if state changes occur in response to the clock signal's *rising* edge (or when it is *high*)
- Similarly, a CLOCK signal is *active low* if state changes occur in response to the clock signal's *falling* edge (or when it is *low*)
- The *clock period* is the time between successive transitions in the same direction
- The *clock frequency* (measured in Hertz, or cycles-per-second) is the *reciprocal* of the clock period
- The *duty cycle* is the percentage of time that the clock signal is at its asserted level

12

Clock Signal Properties



13

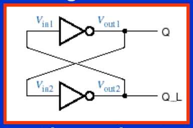
Types of Sequential Circuits

- There are two basic types of sequential circuits that account for the majority of practical discrete designs:
  - a ***feedback sequential circuit*** uses ordinary gates and feedback loops to create sequential circuit building blocks such as latches and flip-flops
  - a ***clocked synchronous state machine*** uses latches and flip-flops (in particular, edge-triggered "D" flip-flops) to create circuits whose inputs are examined and whose outputs change state in accordance with a controlling clock signal

14

Bi-stable Elements

- The "simplest" sequential circuit consists of a pair of inverters forming a feedback loop:



- This element has no inputs and therefore no way of controlling or changing its state
- When power is first applied, it randomly comes up in one state or the other and stays there forever ("not very useful")

15

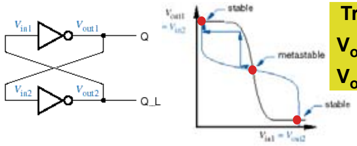
Digital Analysis of Bi-stable

- This circuit is called a ***bi-stable*** because, based on (strictly) digital analysis, it has ***two*** stable states:
  - if Q is HIGH, then the bottom inverter has a high input and a LOW output, which forces the top inverter's output HIGH
  - if Q is LOW, then the bottom inverter has a LOW input and a HIGH output, which forces Q to go LOW
- Based on this analysis, a single state variable ("Q") could be used to describe the state of this circuit

16

Analog Analysis of Bi-stable

- **Given the feedback connection, we know that $V_{in1} = V_{out2}$ and $V_{in2} = V_{out1}$**



Transfer fns:

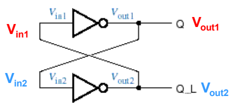
$V_{out1} = T(V_{in1})$

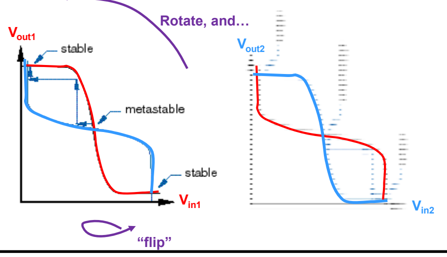
$V_{out2} = T(V_{in2})$

- **The feedback loop is in *equilibrium* if the input and output voltages of both inverters are constant DC values consistent with their transfer functions**

17

Analog Analysis





18

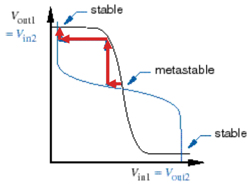
Analog Analysis of Bi-stable

- The *equilibrium points* can be found graphically – they are the points at which the two transfer functions meet:
  - the two *stable* equilibrium points correspond to the two states identified in the "digital" analysis, with Q (Q_L) either "0" (LOW) or "1" (HIGH)
  - the *metastable* equilibrium point occurs with V_{out1} and V_{out2} about halfway between a valid logic "1" voltage and a valid logic "0" voltage – here, Q and Q_L are not valid logic signals but the loop equations are satisfied

19

Metastable Behavior

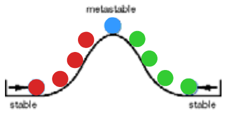
- The *metastable point* is *not truly stable*, because *random noise* will tend to drive a circuit operating at the metastable point toward one of the stable operating points



20

Metastable Behavior

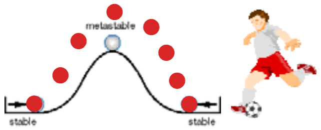
- Metastable behavior of a bistable can be compared to the behavior of a ball dropped onto a hill:
  - if ball is dropped from overhead, it will probably roll down immediately to one side of the hill or the other
  - if ball lands right at the top, it may sit there a while before random forces start it rolling



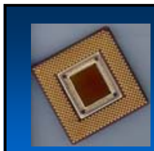
21

Metastable Behavior

- Important: If the "simplest" sequential circuit is susceptible to metastable behavior, you can be sure that *all* sequential circuits are susceptible (and it is not something that only occurs at power-up)
- Consider what happens if we try to "kick" the ball from side of the hill to the other:



22



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 3-B

Set-Reset (S-R) and Data (D) Latches

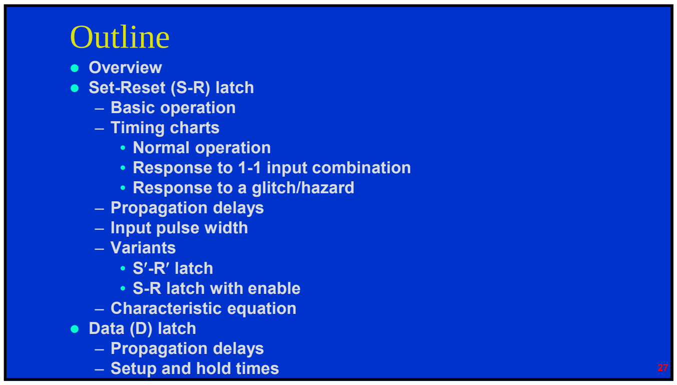
Reading Assignment:

DDPP 4th Ed. pp. 526-532, *DDPP* 5th Ed. pp. 499-504

Learning Objectives:

- Write present state – next state (PS-NS) equations that describes the behavior of a sequential circuit
- Draw a state transition diagram that depicts the behavior of a sequential circuit
- Construct a timing diagram that depicts the behavior of a sequential circuit
- Draw a circuit for a set-reset latch and analyze its behavior
- Discuss what is meant by "transparent" (or "data following") in reference to the response of a latch

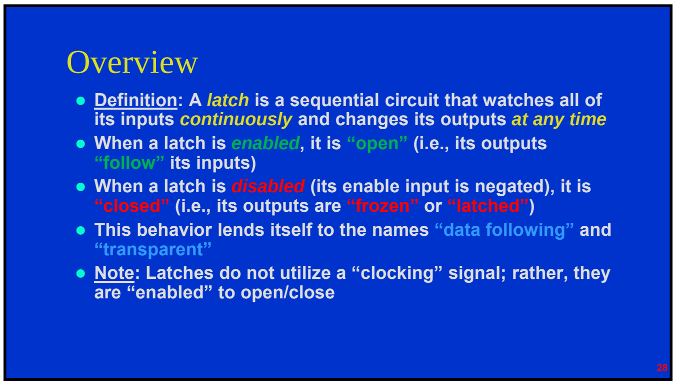
26



Outline

- Overview
- Set-Reset (S-R) latch
  - Basic operation
  - Timing charts
    - Normal operation
    - Response to 1-1 input combination
    - Response to a glitch/hazard
  - Propagation delays
  - Input pulse width
  - Variants
    - S′-R′ latch
    - S-R latch with enable
  - Characteristic equation
- Data (D) latch
  - Propagation delays
  - Setup and hold times

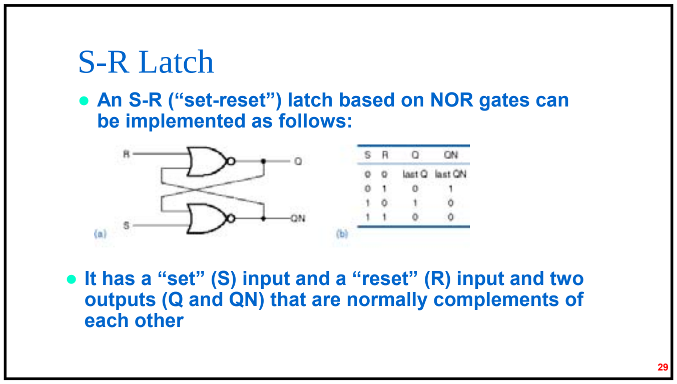
27



Overview

- Definition: A *latch* is a sequential circuit that watches all of its inputs *continuously* and changes its outputs *at any time*
- When a latch is *enabled*, it is "open" (i.e., its outputs "follow" its inputs)
- When a latch is *disabled* (its enable input is negated), it is "closed" (i.e., its outputs are "frozen" or "latched")
- This behavior lends itself to the names "data following" and "transparent"
- Note: Latches do not utilize a "clocking" signal; rather, they are "enabled" to open/close

28



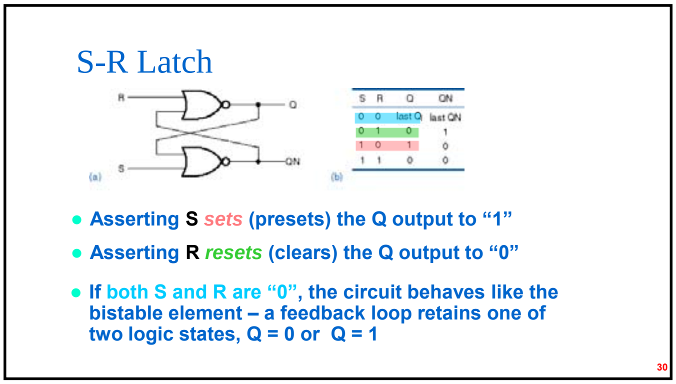
S-R Latch

- An S-R ("set-reset") latch based on NOR gates can be implemented as follows:

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

- It has a "set" (S) input and a "reset" (R) input and two outputs (Q and QN) that are normally complements of each other

29

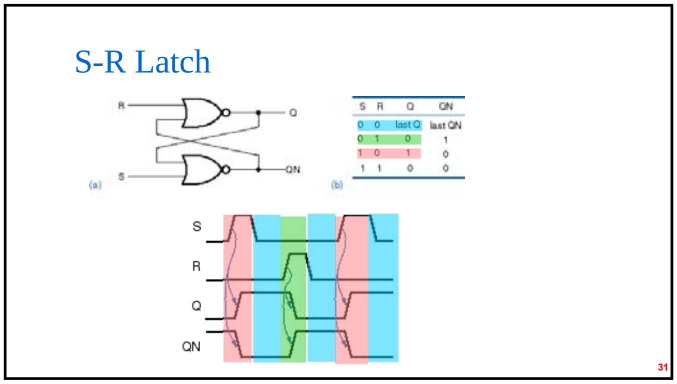


S-R Latch

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

- Asserting **S** *sets* (presets) the Q output to "1"
- Asserting **R** *resets* (clears) the Q output to "0"
- If both S and R are "0", the circuit behaves like the bistable element – a feedback loop retains one of two logic states, Q = 0 or Q = 1

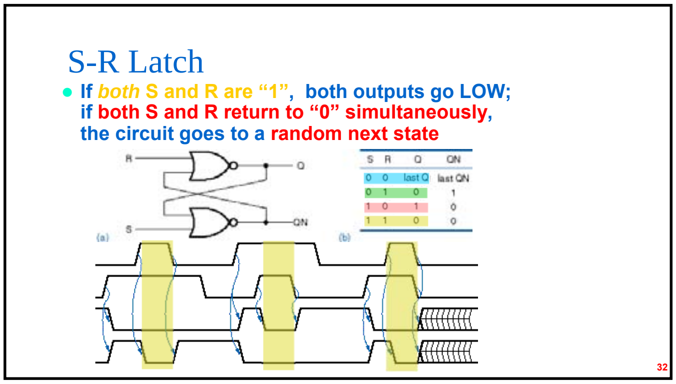
30



S-R Latch

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

31

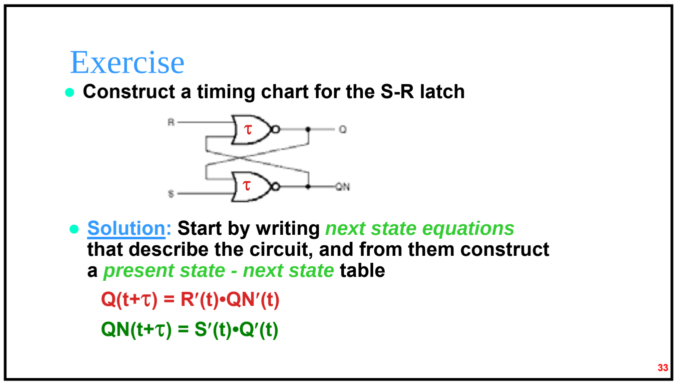


S-R Latch

- If *both* S and R are "1", both outputs go LOW; if both S and R return to "0" simultaneously, the circuit goes to a random next state

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

32



Exercise

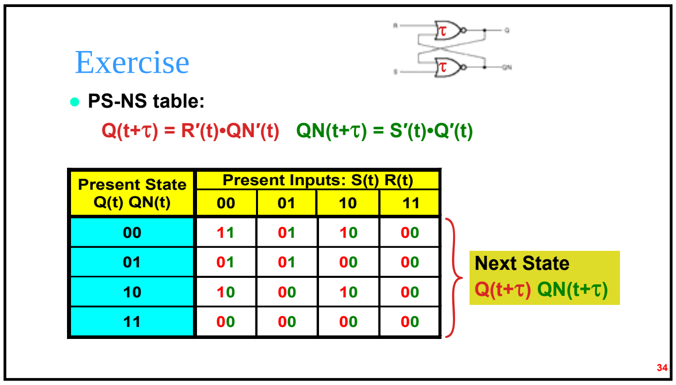
- Construct a timing chart for the S-R latch

- Solution: Start by writing *next state equations* that describe the circuit, and from them construct a *present state - next state* table

  $Q(t+\tau) = R'(t) \bullet QN'(t)$

  $QN(t+\tau) = S'(t) \bullet Q'(t)$

33



Exercise

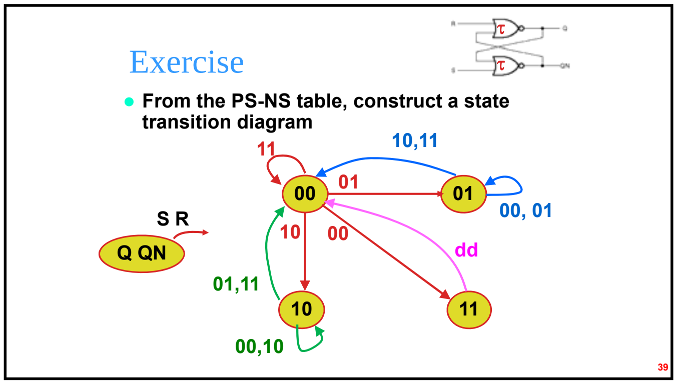
- PS-NS table:

  $Q(t+\tau) = R'(t) \bullet QN'(t)$ $QN(t+\tau) = S'(t) \bullet Q'(t)$

Present State Q(t) QN(t)	Present Inputs: S(t) R(t)			
	00	01	10	11
00	11	01	10	00
01	01	01	00	00
10	10	00	10	00
11	00	00	00	00

Next State
Q(t+τ) QN(t+τ)

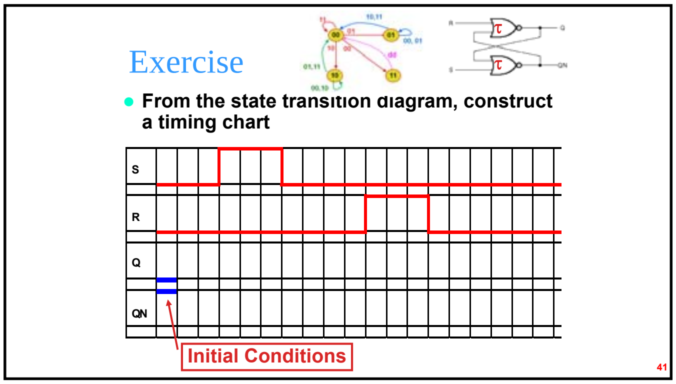
34



Exercise

- From the PS-NS table, construct a state transition diagram

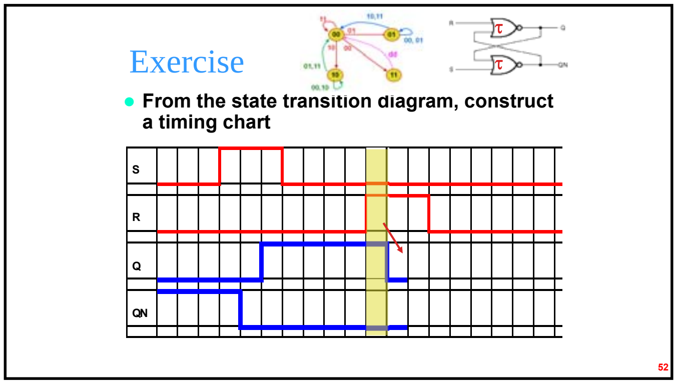
39



Exercise

- From the state transition diagram, construct a timing chart

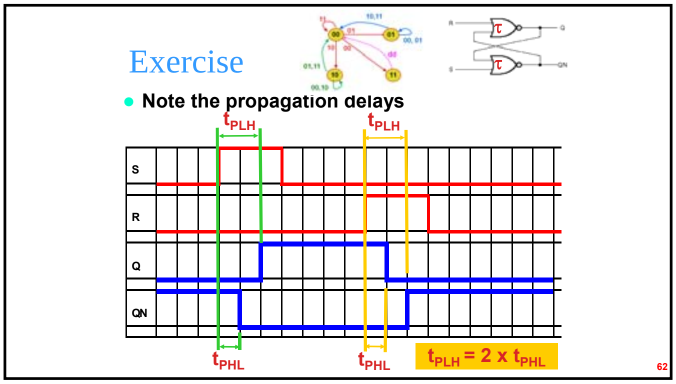
41



Exercise

- From the state transition diagram, construct a timing chart

52

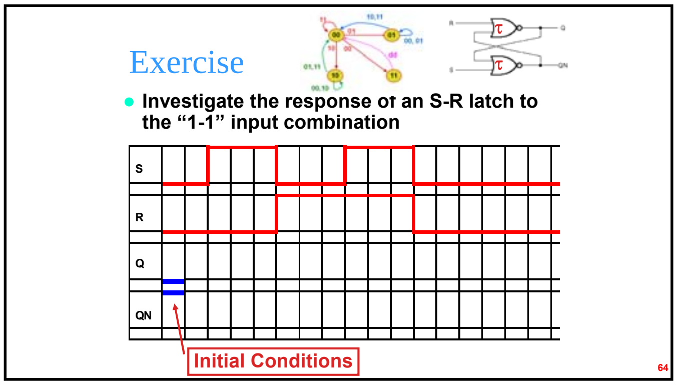


Exercise

- Note the propagation delays

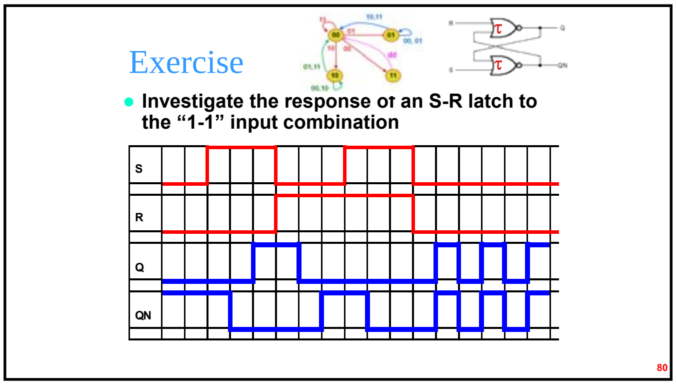
$t_{PLH} = 2 \times t_{PHL}$

62



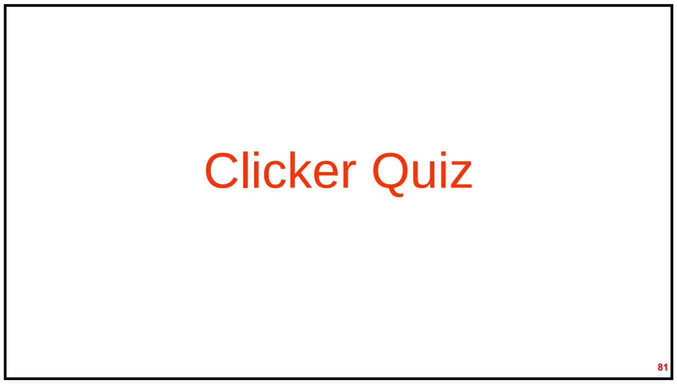
Exercise

- Investigate the response of an S-R latch to the “1-1” input combination

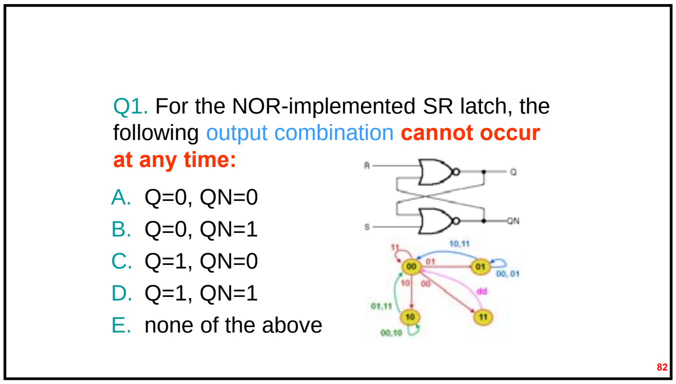


Exercise

- Investigate the response of an S-R latch to the “1-1” input combination

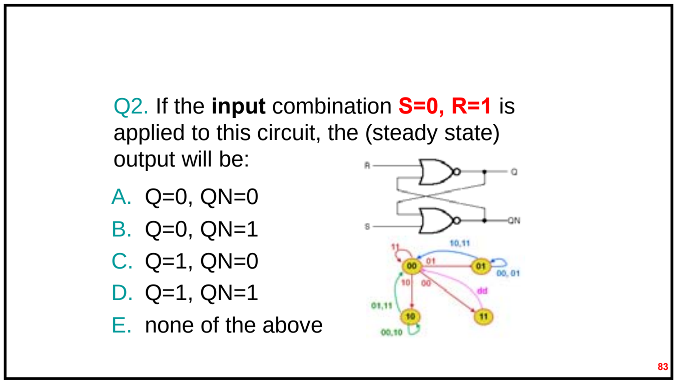


Clicker Quiz



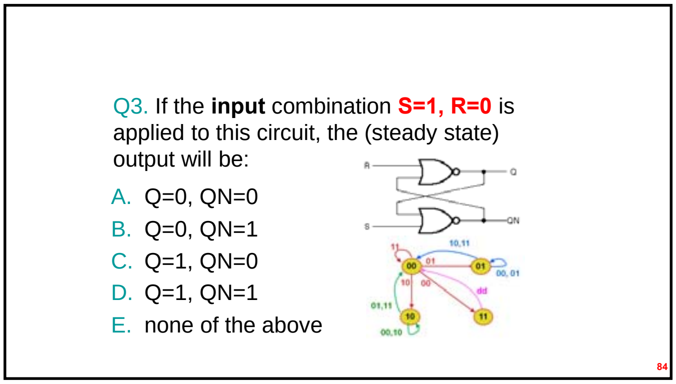
Q1. For the NOR-implemented SR latch, the following output combination **cannot occur at any time:**

A. Q=0, QN=0
B. Q=0, QN=1
C. Q=1, QN=0
D. Q=1, QN=1
E. none of the above



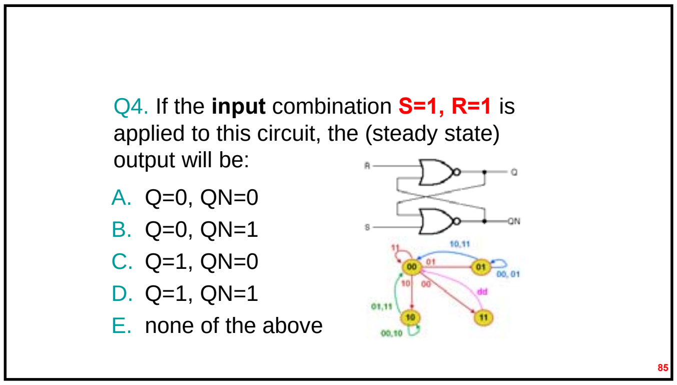
Q2. If the **input** combination **S=0, R=1** is applied to this circuit, the (steady state) output will be:

A. Q=0, QN=0
B. Q=0, QN=1
C. Q=1, QN=0
D. Q=1, QN=1
E. none of the above



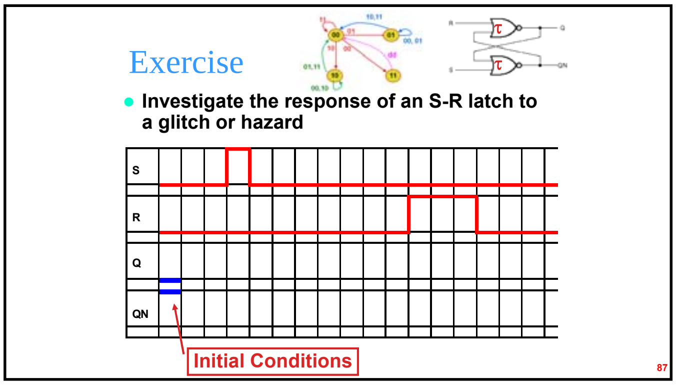
Q3. If the **input** combination **S=1, R=0** is applied to this circuit, the (steady state) output will be:

A. Q=0, QN=0
B. Q=0, QN=1
C. Q=1, QN=0
D. Q=1, QN=1
E. none of the above



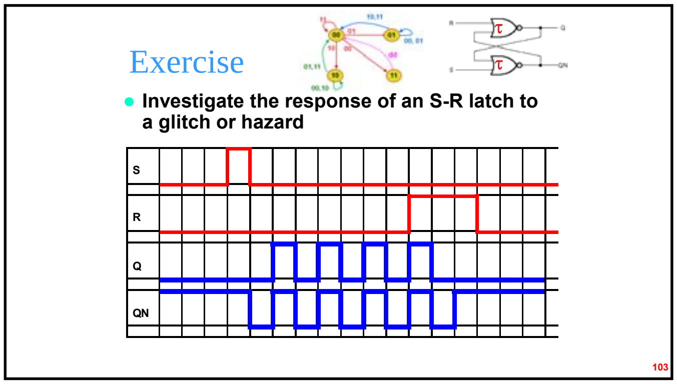
Q4. If the **input** combination **S=1, R=1** is applied to this circuit, the (steady state) output will be:

A. Q=0, QN=0
B. Q=0, QN=1
C. Q=1, QN=0
D. Q=1, QN=1
E. none of the above



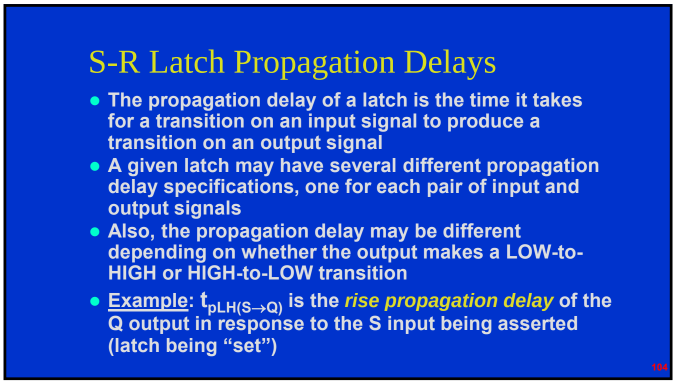
Exercise

- Investigate the response of an S-R latch to a glitch or hazard



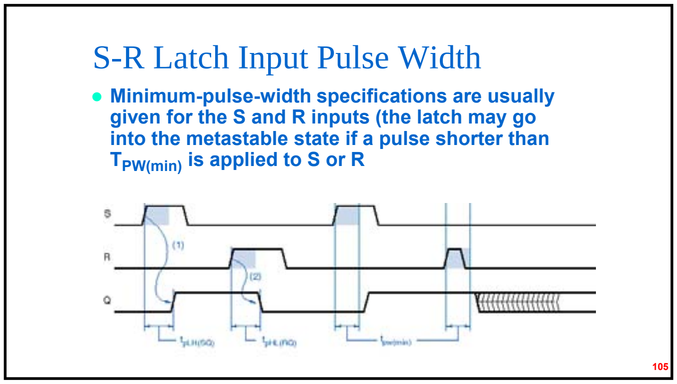
Exercise

- Investigate the response of an S-R latch to a glitch or hazard



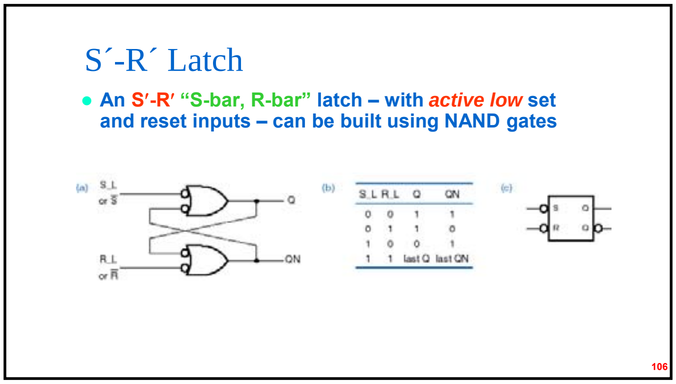
S-R Latch Propagation Delays

- The propagation delay of a latch is the time it takes for a transition on an input signal to produce a transition on an output signal
- A given latch may have several different propagation delay specifications, one for each pair of input and output signals
- Also, the propagation delay may be different depending on whether the output makes a LOW-to-HIGH or HIGH-to-LOW transition
- Example: $t_{pLH(S \rightarrow Q)}$ is the *rise propagation delay* of the Q output in response to the S input being asserted (latch being "set")



S-R Latch Input Pulse Width

- Minimum-pulse-width specifications are usually given for the S and R inputs (the latch may go into the metastable state if a pulse shorter than $T_{PW(min)}$ is applied to S or R



S´-R´ Latch

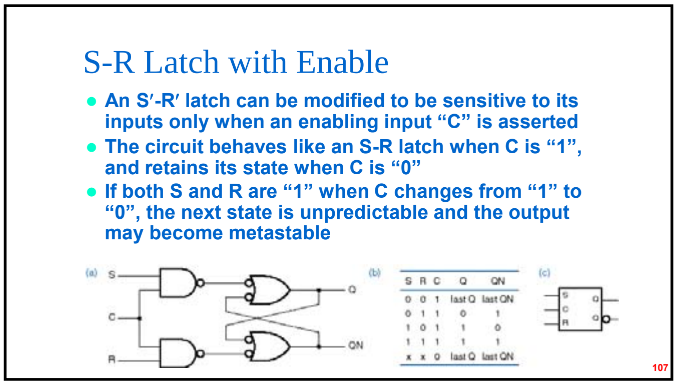
- An S′-R′ "S-bar, R-bar" latch – with *active low* set and reset inputs – can be built using NAND gates

(a)

(b)

S_L	R_L	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	last Q	last QN

(c)

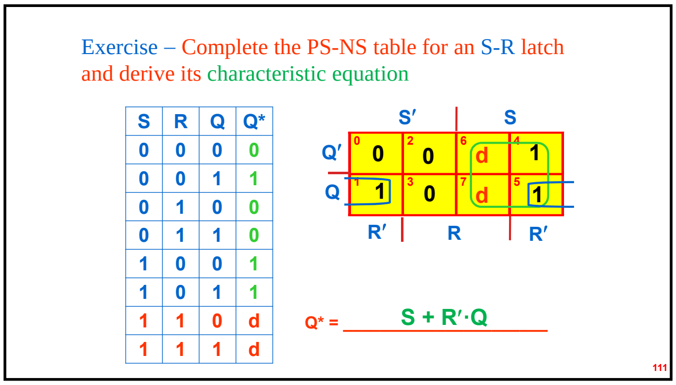


S-R Latch with Enable

- An S′-R′ latch can be modified to be sensitive to its inputs only when an enabling input “C” is asserted
- The circuit behaves like an S-R latch when C is “1”, and retains its state when C is “0”
- If both S and R are “1” when C changes from “1” to “0”, the next state is unpredictable and the output may become metastable

S	R	C	Q	QN
0	0	1	last Q	last QN
0	1	1	0	1
1	0	1	1	0
1	1	1	1	1
x	x	0	last Q	last QN

107



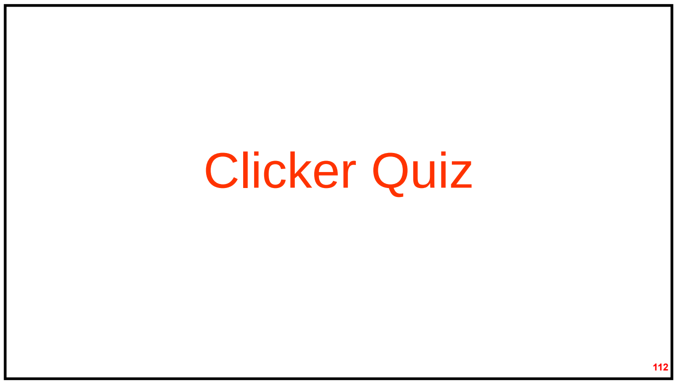
Exercise – Complete the PS-NS table for an S-R latch and derive its characteristic equation

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	d
1	1	1	d

	R′	R	R	R′
	S′	S′	S	S
Q′	^{0} 0	^{2} 0	^{6} d	^{4} 1
Q	^{1} 1	^{3} 0	^{7} d	^{5} 1

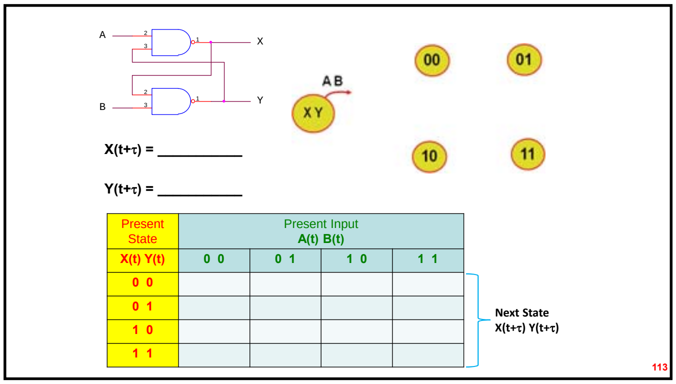
Q* = S + R′·Q

111



Clicker Quiz

112



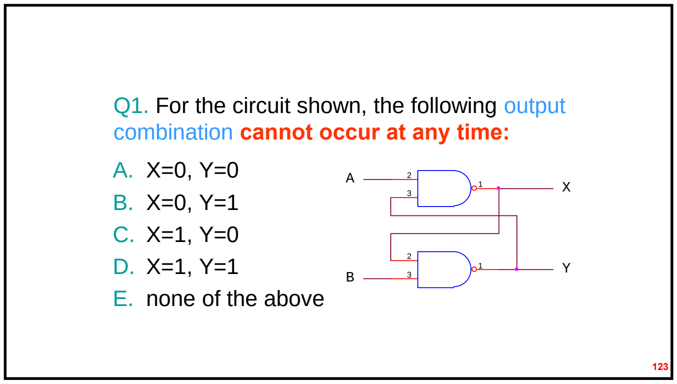
X(t+τ) = ___________

Y(t+τ) = ___________

Present State	Present Input A(t) B(t)			
X(t) Y(t)	0 0	0 1	1 0	1 1
0 0				
0 1				
1 0				
1 1				

Next State
X(t+τ) Y(t+τ)

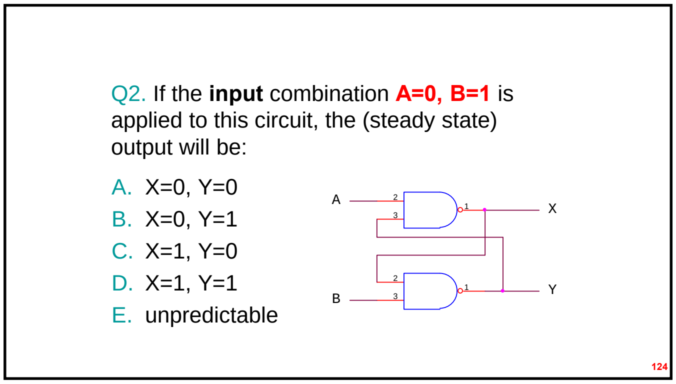
113



Q1. For the circuit shown, the following output combination **cannot occur at any time:**

A. X=0, Y=0
B. X=0, Y=1
C. X=1, Y=0
D. X=1, Y=1
E. none of the above

123



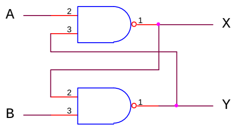
Q2. If the **input** combination **A=0, B=1** is applied to this circuit, the (steady state) output will be:

A. X=0, Y=0
B. X=0, Y=1
C. X=1, Y=0
D. X=1, Y=1
E. unpredictable

124

Q3. If the **input** combination **A=1, B=0** is applied to this circuit, the (steady state) output will be:

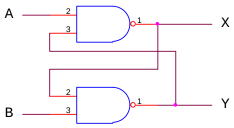
A. X=0, Y=0
B. X=0, Y=1
C. X=1, Y=0
D. X=1, Y=1
E. unpredictable



125

Q4. If the **input** combination **A=0, B=0** is applied to this circuit, **followed immediately** by the **input** combination **A=1, B=1,** the (steady state) output will be:

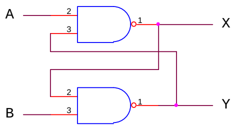
A. X=0, Y=0
B. X=0, Y=1
C. X=1, Y=0
D. X=1, Y=1
E. unpredictable



126

Q5. If the **propagation delay** of each gate is **10 ns,** the **minimum length of time** that (valid) input combinations need to be asserted **in order to prevent metastable behavior** is:

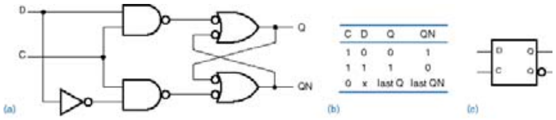
A. 10 ns
B. 20 ns
C. 30 ns
D. 40 ns
E. none of the above



127

Transparent D Latch

- **In situations where we simply need to store a single "bit" of information, a *D ("data") latch* can be used**



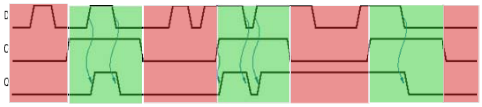
C	D	Q	QN
1	0	0	1
1	1	1	0
0	x	last Q	last QN

- **Note that a D latch is just an S-R latch, with D connected to the S input and D′ connected to the R input (this eliminates the troublesome "1-1" input combination)**

128

Transparent D Latch

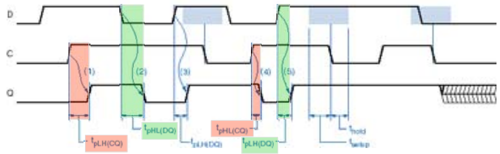
- **When the enable input C is asserted, the latch is said to be "open" and the path from the D input to the Q output is "transparent" – hence the name *transparent latch***
- **When the enable input C is negated, the latch "closes" – the Q output retains its last value and no longer changes in response to D**



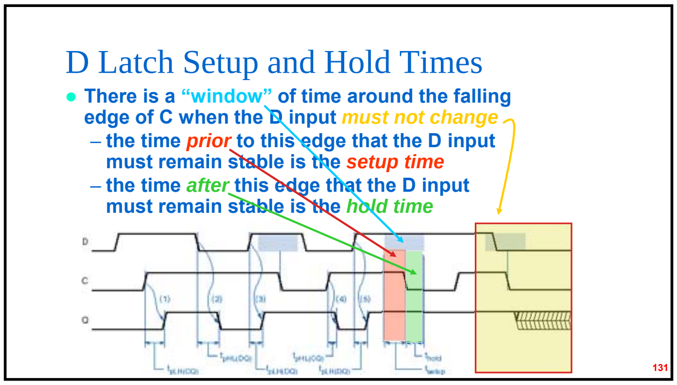
129

D Latch Propagation Delays

- **There are four propagation delay parameters that must be considered:**
  - $t_{pLH(C \rightarrow Q)}$ and $t_{pHL(C \rightarrow Q)}$
  - $t_{pLH(D \rightarrow Q)}$ and $t_{pHL(D \rightarrow Q)}$



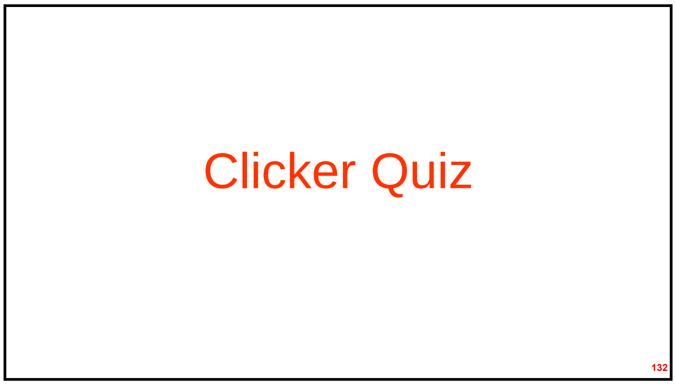
130



D Latch Setup and Hold Times

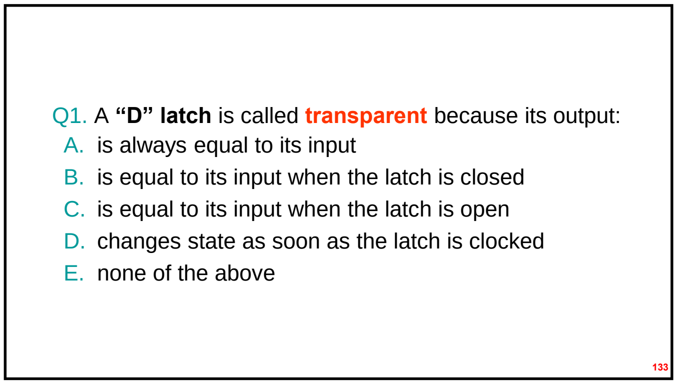
- There is a "window" of time around the falling edge of C when the D input *must not change*
  - the time *prior* to this edge that the D input must remain stable is the *setup time*
  - the time *after* this edge that the D input must remain stable is the *hold time*

131



Clicker Quiz

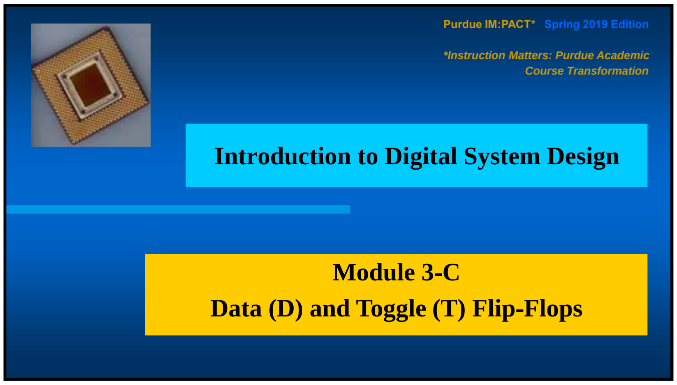
132



Q1. A **"D" latch** is called **transparent** because its output:

A. is always equal to its input
B. is equal to its input when the latch is closed
C. is equal to its input when the latch is open
D. changes state as soon as the latch is clocked
E. none of the above

133

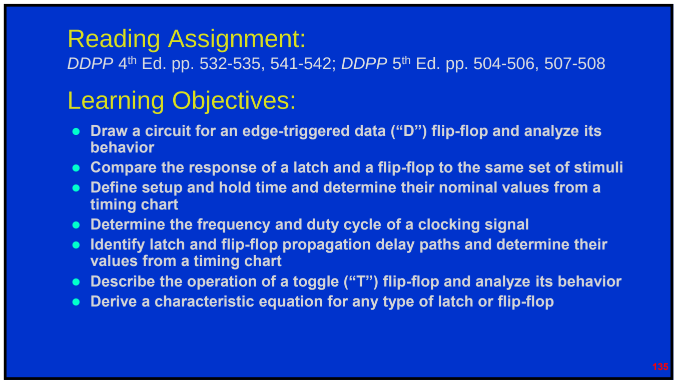


Purdue IM:PACT* Spring 2019 Edition

**Instruction Matters: Purdue Academic Course Transformation*

Introduction to Digital System Design

Module 3-C
Data (D) and Toggle (T) Flip-Flops



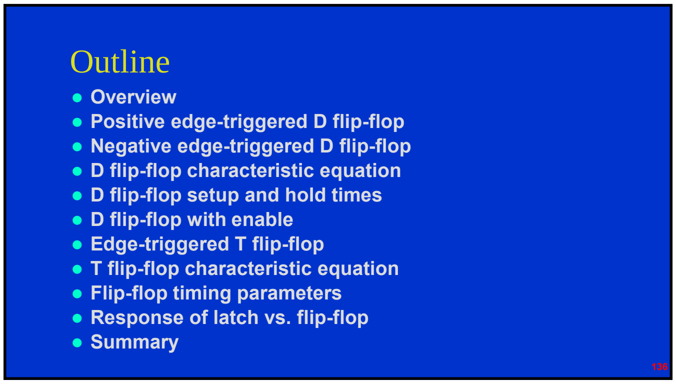
Reading Assignment:

DDPP 4th Ed. pp. 532-535, 541-542; *DDPP* 5th Ed. pp. 504-506, 507-508

Learning Objectives:

- Draw a circuit for an edge-triggered data ("D") flip-flop and analyze its behavior
- Compare the response of a latch and a flip-flop to the same set of stimuli
- Define setup and hold time and determine their nominal values from a timing chart
- Determine the frequency and duty cycle of a clocking signal
- Identify latch and flip-flop propagation delay paths and determine their values from a timing chart
- Describe the operation of a toggle ("T") flip-flop and analyze its behavior
- Derive a characteristic equation for any type of latch or flip-flop

135



Outline

- Overview
- Positive edge-triggered D flip-flop
- Negative edge-triggered D flip-flop
- D flip-flop characteristic equation
- D flip-flop setup and hold times
- D flip-flop with enable
- Edge-triggered T flip-flop
- T flip-flop characteristic equation
- Flip-flop timing parameters
- Response of latch vs. flip-flop
- Summary

136

Overview

- Definition: A *flip-flop* is a sequential circuit that samples its inputs and *changes its outputs* only at times determined by a *clocking signal* (“CLK”)
- Flip-flops change state in response to the *transition* (“edge”) of a clocking signal
  - *positive-edge-triggered* flip-flops change state on the *low-to-high* transition of a clocking signal
  - *negative-edge-triggered* flip-flops change state on the *high-to-low* transition of a clocking signal

Positive Edge-Triggered D Flip-Flop

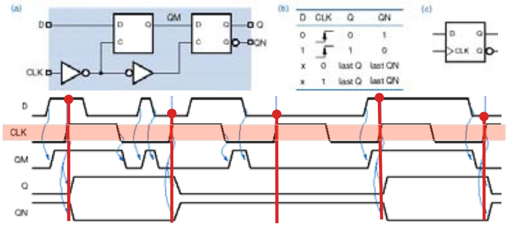
- A *positive-edge-triggered D flip flop* combines a pair of D latches to create a circuit that samples its D input and changes its Q and QN outputs *at the rising edge* of a controlling CLOCK (CLK) signal
  - the first latch, called the *master*, opens and follows the input when CLK is 0
  - the second latch, called the *slave*, opens and reads the master’s output when CLK is 1 – this is when the output state change occurs (note that the master latch is *closed* at this point and thus “immune” to input changes)

Positive Edge-Triggered D Flip-Flop

- A *triangle* on the D flip-flop’s CLK input indicates edge-triggered behavior and is called a *dynamic input indicator*
- The characteristic equation of a D flip-flop is Q* = D – i.e. the next state is the current input, shorthand for Q(t+Δ) = D(t), where Δ is the *clocking period*
- D flip-flops are included in the macrocells of virtually all PLDs, and are therefore the “most popular” (and easiest) way to realize clocked synchronous state machines

Positive Edge-Triggered D Flip-Flop

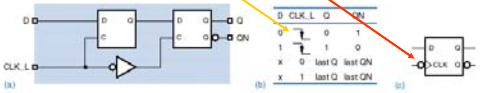
- One way an edge-triggered D flip flop can be constructed is illustrated below



D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

Negative Edge-Triggered D Flip-Flop

- D flip flops can also be designed to be *negative-edge-triggered*
- An *inversion bubble* on the CLK input is used to indicate that a flip flop is triggered on the HIGH-to-LOW transition of the CLK signal



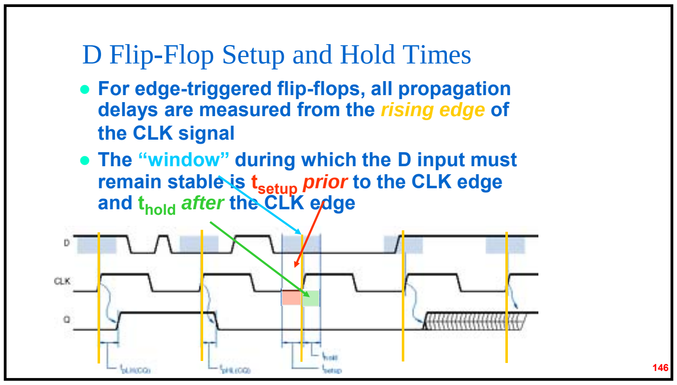
D	CLK_L	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN

Exercise – Complete the PS-NS table for an D flip-flop and derive its characteristic equation

D	Q	Q*
0	0	0
0	1	0
1	0	1
1	1	1

	D′	D
Q′	^{0} 0	^{2} 1
Q	^{1} 0	^{3} 1

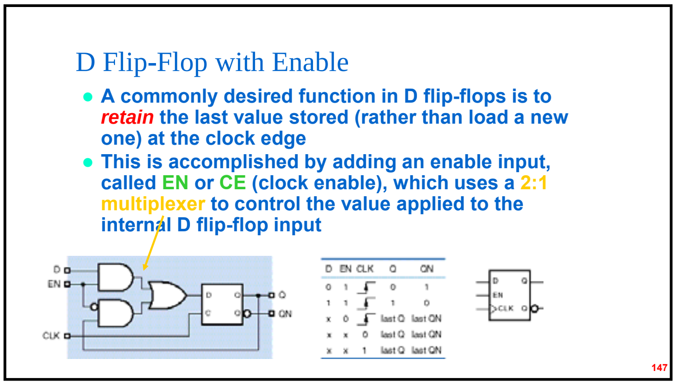
Q* = D



D Flip-Flop Setup and Hold Times

- For edge-triggered flip-flops, all propagation delays are measured from the *rising edge* of the CLK signal
- The "window" during which the D input must remain stable is t_{setup} *prior* to the CLK edge and t_{hold} *after* the CLK edge

146

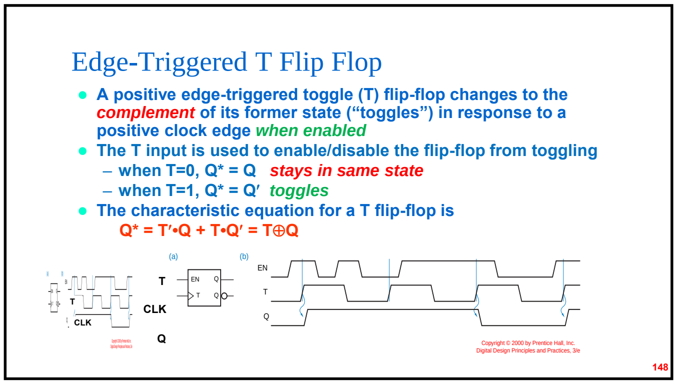


D Flip-Flop with Enable

- A commonly desired function in D flip-flops is to *retain* the last value stored (rather than load a new one) at the clock edge
- This is accomplished by adding an enable input, called EN or CE (clock enable), which uses a 2:1 multiplexer to control the value applied to the internal D flip-flop input

D	EN	CLK	Q	QN
0	1	↑	0	1
1	1	↑	1	0
x	0	↑	last Q	last QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN

147



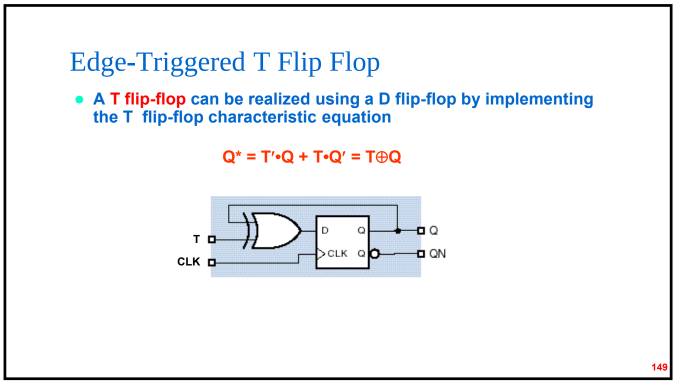
Edge-Triggered T Flip Flop

- A positive edge-triggered toggle (T) flip-flop changes to the *complement* of its former state ("toggles") in response to a positive clock edge *when enabled*
- The T input is used to enable/disable the flip-flop from toggling
  - when T=0, Q* = Q *stays in same state*
  - when T=1, Q* = Q′ *toggles*
- The characteristic equation for a T flip-flop is

  $Q^* = T'\bullet Q + T\bullet Q' = T\oplus Q$

Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

148

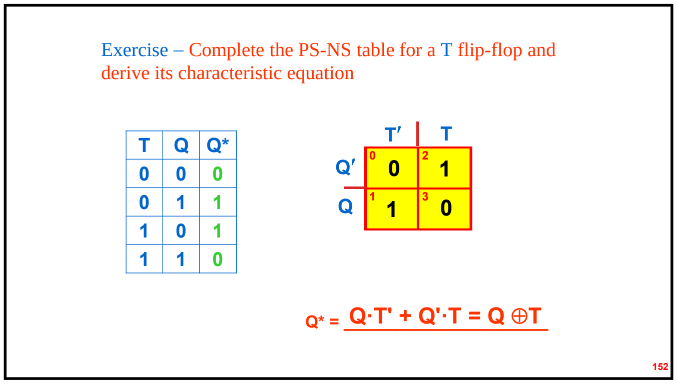


Edge-Triggered T Flip Flop

- A T flip-flop can be realized using a D flip-flop by implementing the T flip-flop characteristic equation

$Q^* = T'\bullet Q + T\bullet Q' = T\oplus Q$

149



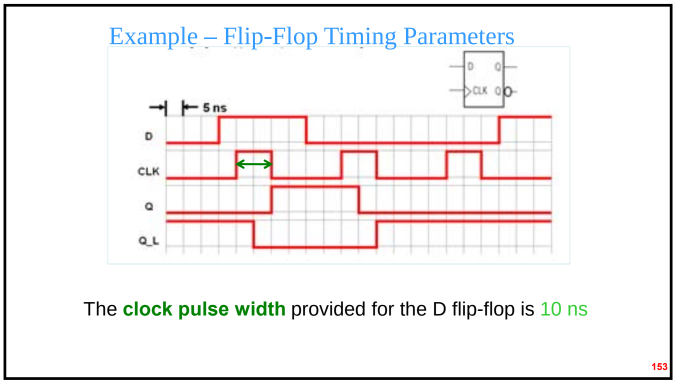
Exercise – Complete the PS-NS table for a T flip-flop and derive its characteristic equation

T	Q	Q*
0	0	0
0	1	1
1	0	1
1	1	0

	T′	T
Q′	0: 0	2: 1
Q	1: 1	3: 0

$Q^* = \underline{Q\cdot T' + Q'\cdot T = Q \oplus T}$

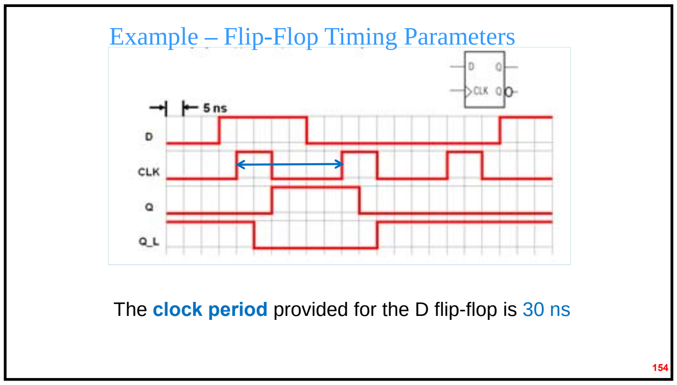
152



Example – Flip-Flop Timing Parameters

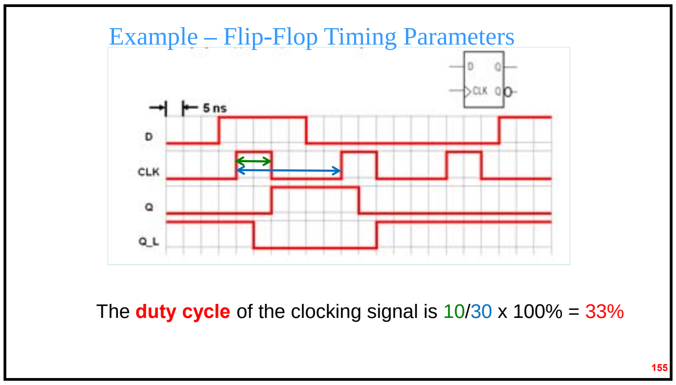
The **clock pulse width** provided for the D flip-flop is 10 ns

153



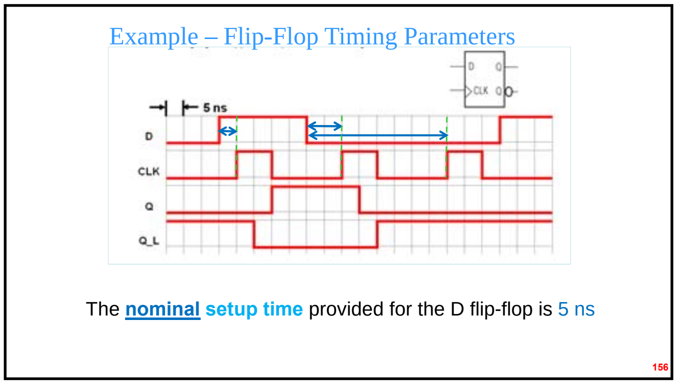
Example – Flip-Flop Timing Parameters

The **clock period** provided for the D flip-flop is 30 ns



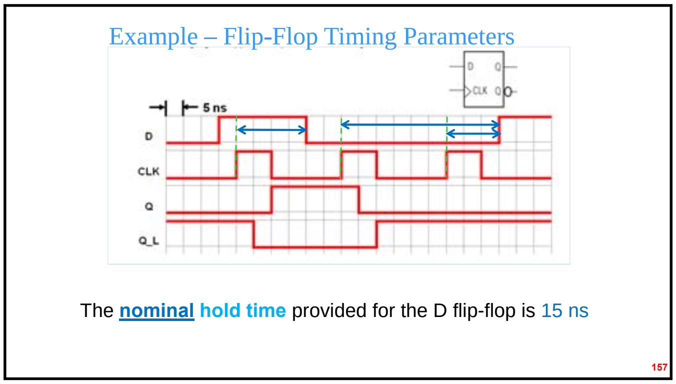
Example – Flip-Flop Timing Parameters

The **duty cycle** of the clocking signal is 10/30 x 100% = 33%



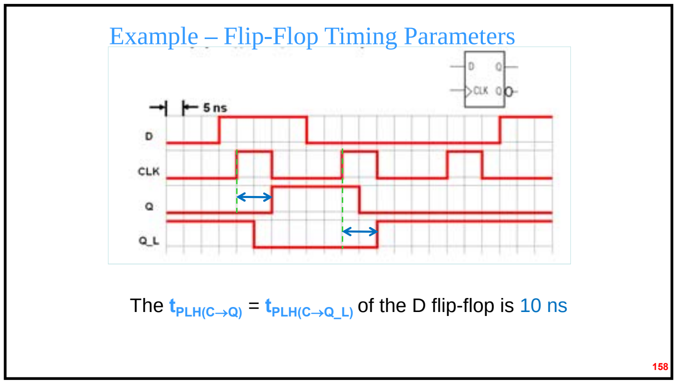
Example – Flip-Flop Timing Parameters

The **nominal** **setup time** provided for the D flip-flop is 5 ns



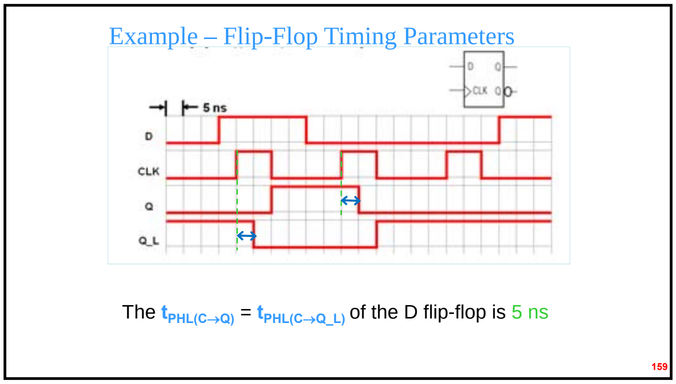
Example – Flip-Flop Timing Parameters

The **nominal** **hold time** provided for the D flip-flop is 15 ns



Example – Flip-Flop Timing Parameters

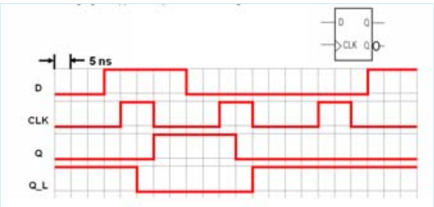
The $t_{PLH(C \rightarrow Q)}$ = $t_{PLH(C \rightarrow Q\_L)}$ of the D flip-flop is 10 ns



Example – Flip-Flop Timing Parameters

The $t_{PHL(C \rightarrow Q)}$ = $t_{PHL(C \rightarrow Q\_L)}$ of the D flip-flop is 5 ns

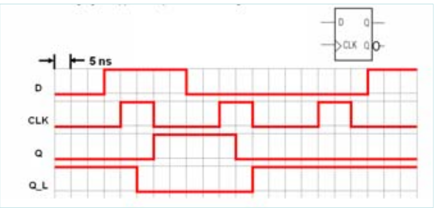
Clicker Quiz



Q1. The **duty cycle** of the clocking signal is:

A. 20% B. 33% C. 40% D. 67%

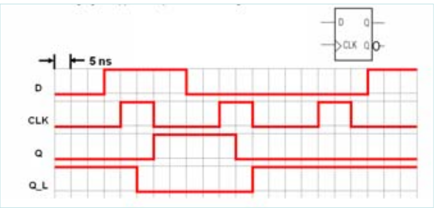
E. none of the above



Q2. The **nominal setup time** provided for the D flip-flop is:

A. 5 ns B. 10 ns C. 15 ns D. 20 ns

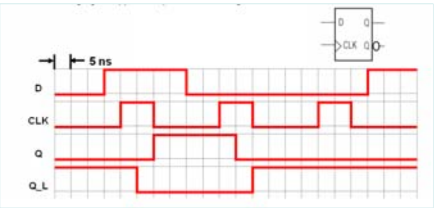
E. none of the above



Q3. The **nominal hold time** provided for the D flip-flop is:

A. 5 ns B. 10 ns C. 15 ns D. 20 ns

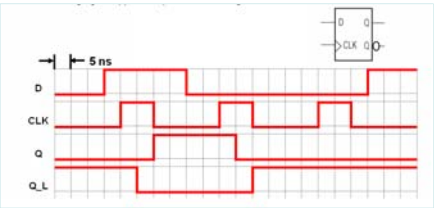
E. none of the above



Q4. The **clock pulse width** provided for the D flip-flop is:

A. 5 ns B. 10 ns C. 15 ns D. 20 ns

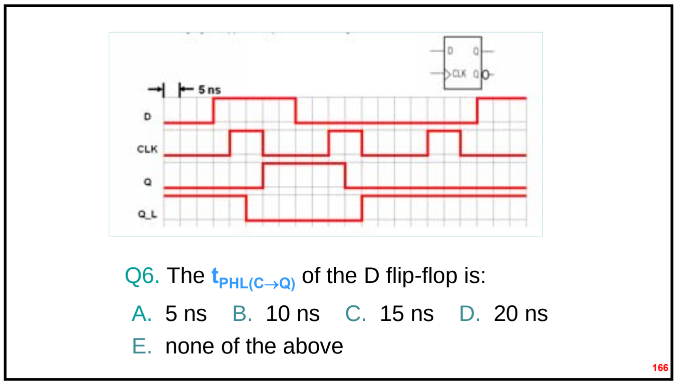
E. none of the above



Q5. The $t_{PLH(C\rightarrow Q)}$ of the D flip-flop is:

A. 5 ns B. 10 ns C. 15 ns D. 20 ns

E. none of the above

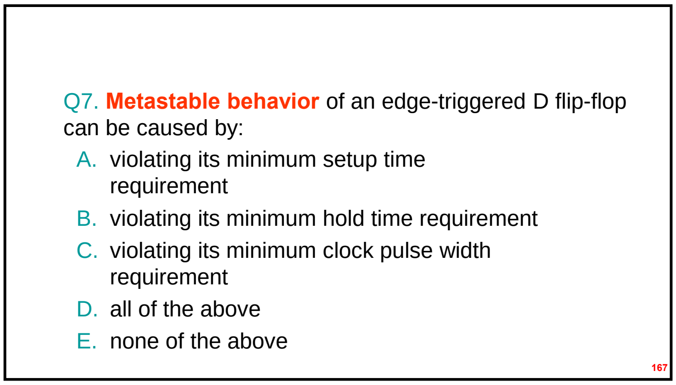


Q6. The $t_{PHL(C \rightarrow Q)}$ of the D flip-flop is:

A. 5 ns B. 10 ns C. 15 ns D. 20 ns

E. none of the above

166



Q7. **Metastable behavior** of an edge-triggered D flip-flop can be caused by:

A. violating its minimum setup time requirement

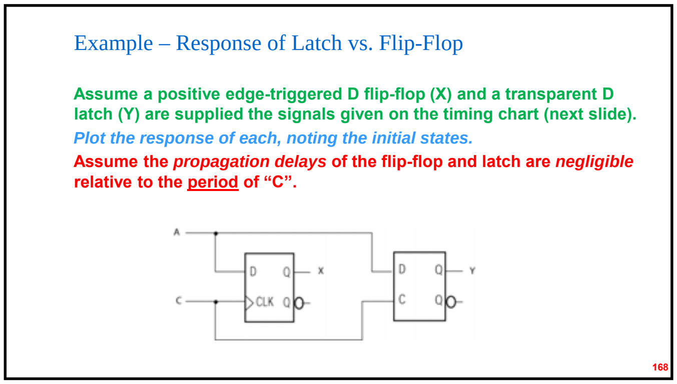
B. violating its minimum hold time requirement

C. violating its minimum clock pulse width requirement

D. all of the above

E. none of the above

167



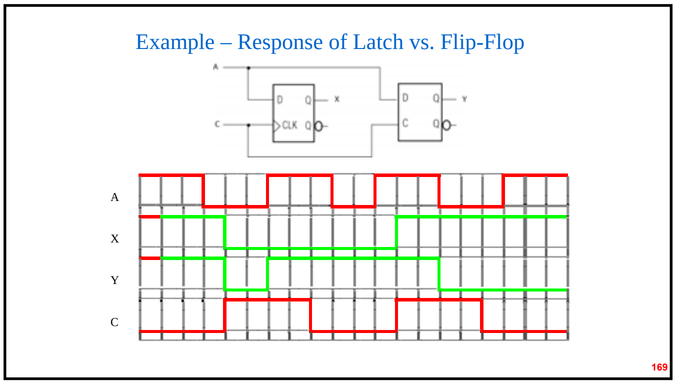
Example – Response of Latch vs. Flip-Flop

Assume a positive edge-triggered D flip-flop (X) and a transparent D latch (Y) are supplied the signals given on the timing chart (next slide).

Plot the response of each, noting the initial states.

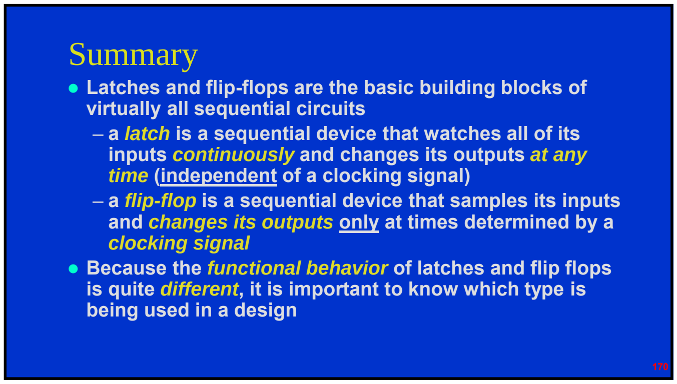
Assume the *propagation delays* of the flip-flop and latch are *negligible* relative to the period of "C".

168



Example – Response of Latch vs. Flip-Flop

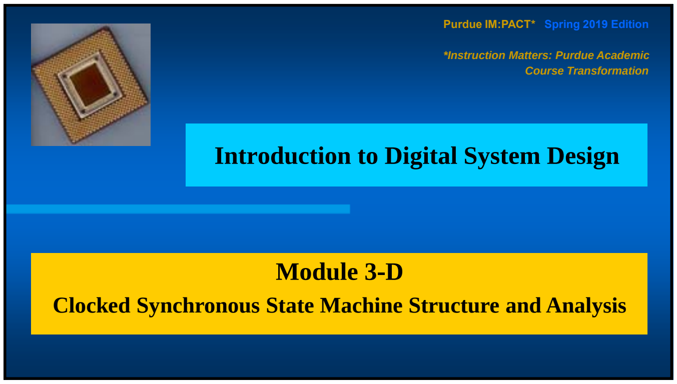
169



Summary

- Latches and flip-flops are the basic building blocks of virtually all sequential circuits
  - a *latch* is a sequential device that watches all of its inputs *continuously* and changes its outputs *at any time* (independent of a clocking signal)
  - a *flip-flop* is a sequential device that samples its inputs and *changes its outputs* only at times determined by a *clocking signal*
- Because the *functional behavior* of latches and flip flops is quite *different*, it is important to know which type is being used in a design

170



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 3-D

Clocked Synchronous State Machine Structure and Analysis

Reading Assignment:
DDPP 4th Ed. pp. 542-553, *DDPP* 5th Ed. pp. 443-453

Learning Objectives:

- Identify the key elements of a clocked synchronous state machine: next state logic, state memory (flip-flops), and output logic
- Differentiate between Mealy and Moore model state machines, and draw a block diagram of each
- Analyze a clocked synchronous state machine realized as either a Mealy or Moore model

172

Outline

- Overview
- State machine structure
  - Moore machine
  - Mealy machine
- State machine analysis
  - Moore machine analysis
  - Mealy machine analysis

173

Overview

- **"State machine"** (or **"finite state machine"**) is a generic name given to **sequential circuits**
- **"Clocked"** indicates that the flip-flops employ a **CLOCK (CLK)** input
- **"Synchronous"** means that all the flip-flops in the state machine use the **same** CLOCK signal
- **"Analysis"** means to **analyze the behavior** of a given state machine
  - construct a PS-NS table
  - derive PS-NS equations
  - draw a state transition diagram
  - draw a timing chart

174

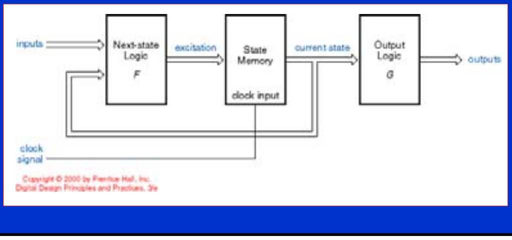
State Machine Structure

- Clocked synchronous state machines consist of three basic blocks:
  - *next state logic* – combinational circuitry that provides the "excitation" necessary to transition to the next state, based on the current state and the present inputs
  - *state memory (flip flops)* – set of N flip-flops that store the current state of the machine (providing 2^N distinct states)
  - *output logic* – combinational circuitry that uses the current state (and possibly current inputs) to determine the outputs generated

175

Moore Machine

- In a *Moore* machine, the outputs are *only* a function of the current state

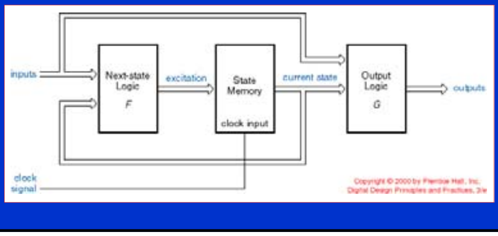


Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

176

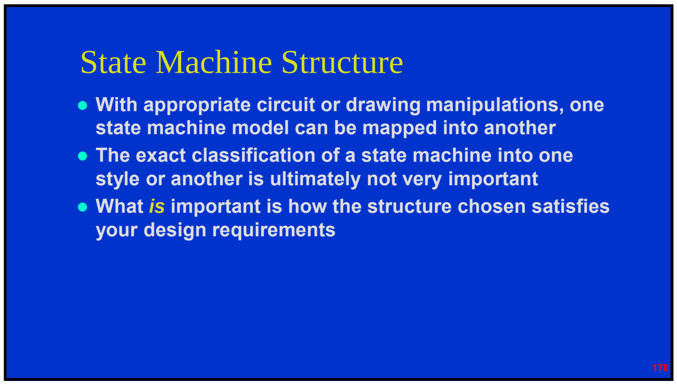
Mealy Machine

- In a *Mealy* machine, the outputs are a function of the current state *as well as* the current inputs



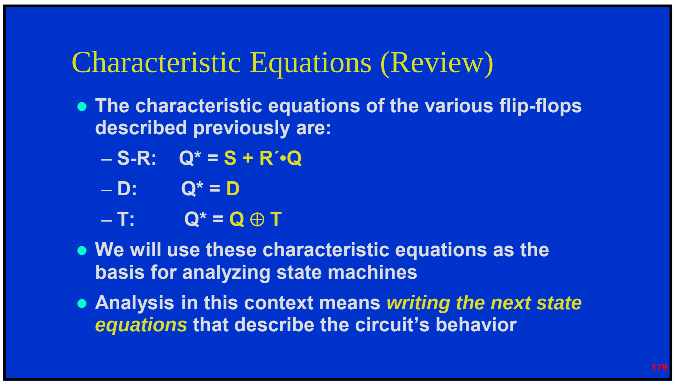
Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

177



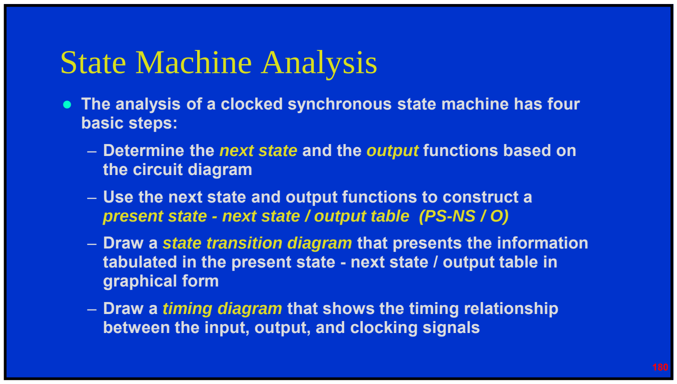
State Machine Structure

- With appropriate circuit or drawing manipulations, one state machine model can be mapped into another
- The exact classification of a state machine into one style or another is ultimately not very important
- What *is* important is how the structure chosen satisfies your design requirements



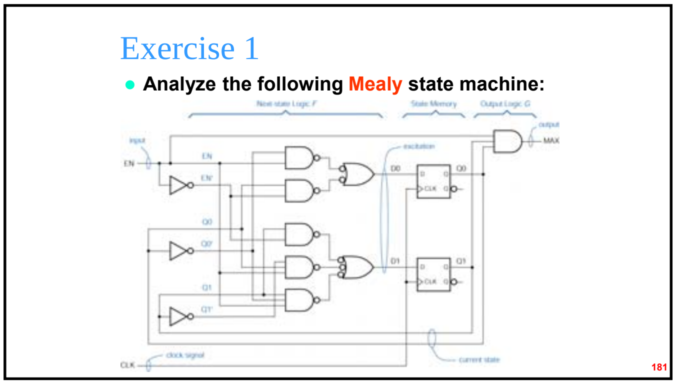
Characteristic Equations (Review)

- The characteristic equations of the various flip-flops described previously are:
  - S-R: $Q^* = S + R' \bullet Q$
  - D: $Q^* = D$
  - T: $Q^* = Q \oplus T$
- We will use these characteristic equations as the basis for analyzing state machines
- Analysis in this context means *writing the next state equations* that describe the circuit's behavior



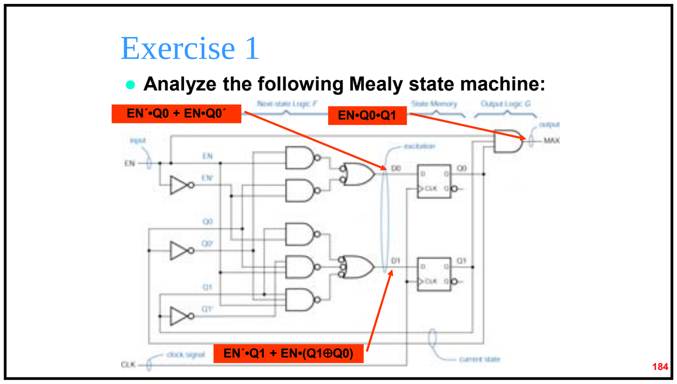
State Machine Analysis

- The analysis of a clocked synchronous state machine has four basic steps:
  - Determine the *next state* and the *output* functions based on the circuit diagram
  - Use the next state and output functions to construct a *present state - next state / output table (PS-NS / O)*
  - Draw a *state transition diagram* that presents the information tabulated in the present state - next state / output table in graphical form
  - Draw a *timing diagram* that shows the timing relationship between the input, output, and clocking signals



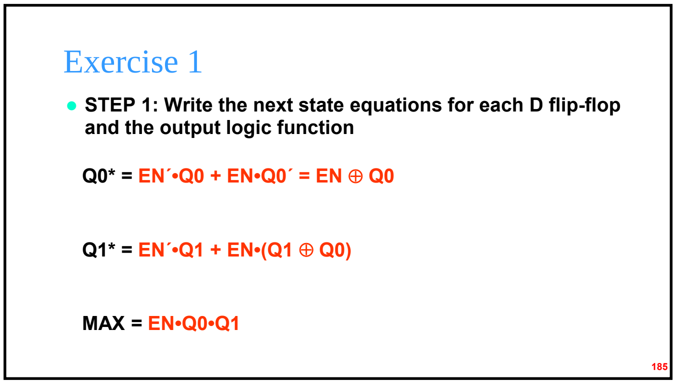
Exercise 1

- Analyze the following Mealy state machine:



Exercise 1

- Analyze the following Mealy state machine:



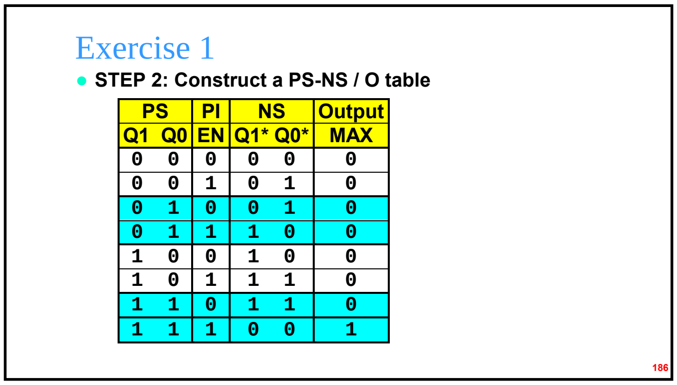
Exercise 1

- STEP 1: Write the next state equations for each D flip-flop and the output logic function

$Q0^* = EN' \bullet Q0 + EN \bullet Q0' = EN \oplus Q0$

$Q1^* = EN' \bullet Q1 + EN \bullet (Q1 \oplus Q0)$

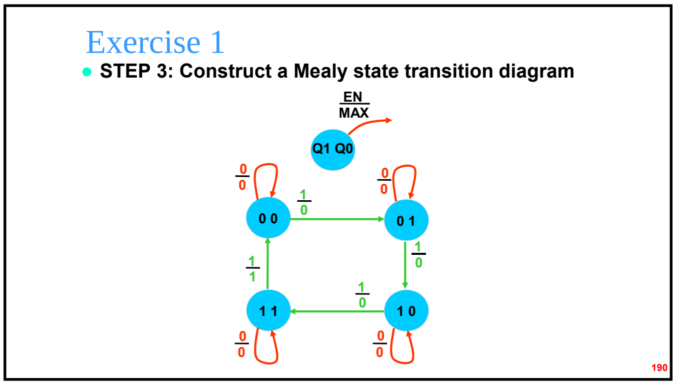
$MAX = EN \bullet Q0 \bullet Q1$



Exercise 1

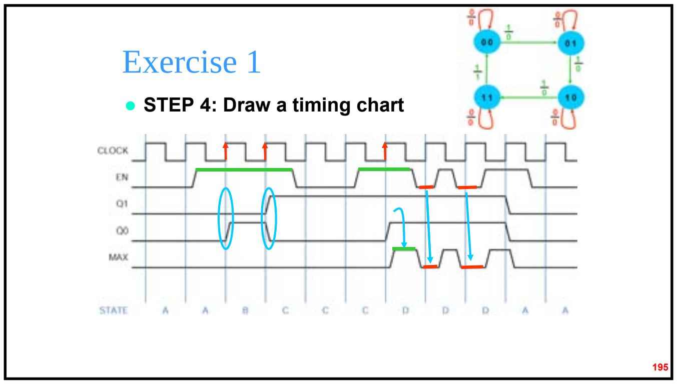
- STEP 2: Construct a PS-NS / O table

PS		PI	NS		Output
Q1	Q0	EN	Q1*	Q0*	MAX
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	0	0	1



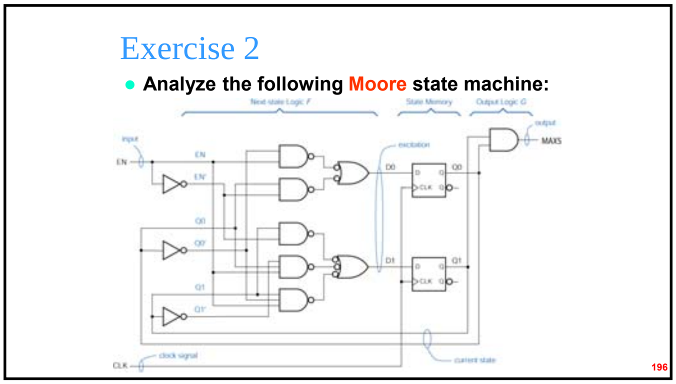
Exercise 1

- STEP 3: Construct a Mealy state transition diagram



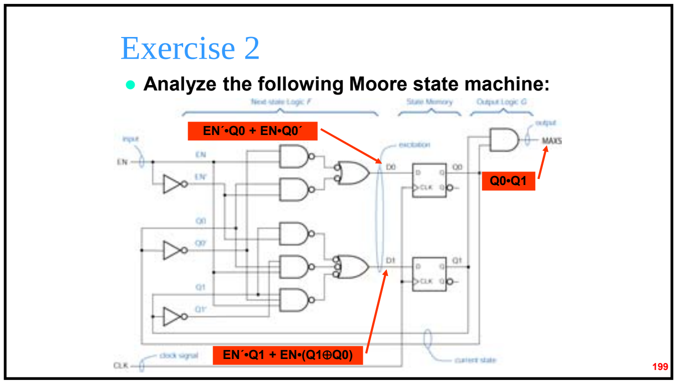
Exercise 1

- STEP 4: Draw a timing chart



Exercise 2

- Analyze the following Moore state machine:



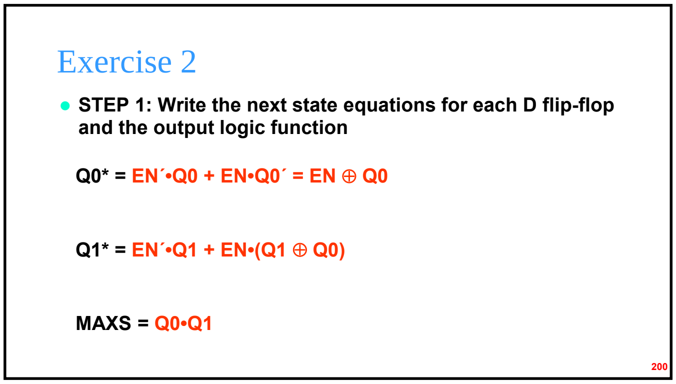
Exercise 2

- Analyze the following Moore state machine:

EN´•Q0 + EN•Q0´

Q0•Q1

EN´•Q1 + EN•(Q1⊕Q0)



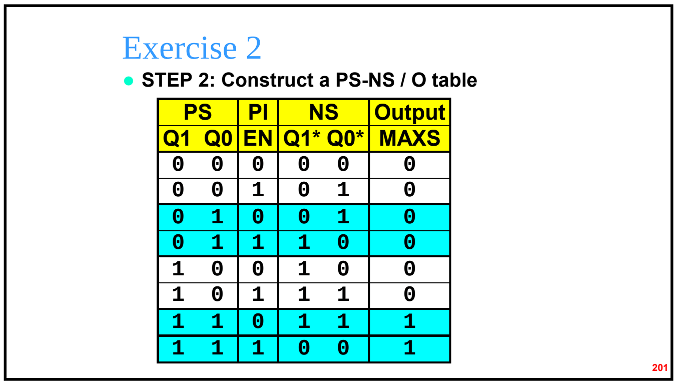
Exercise 2

- STEP 1: Write the next state equations for each D flip-flop and the output logic function

Q0* = EN´•Q0 + EN•Q0´ = EN ⊕ Q0

Q1* = EN´•Q1 + EN•(Q1 ⊕ Q0)

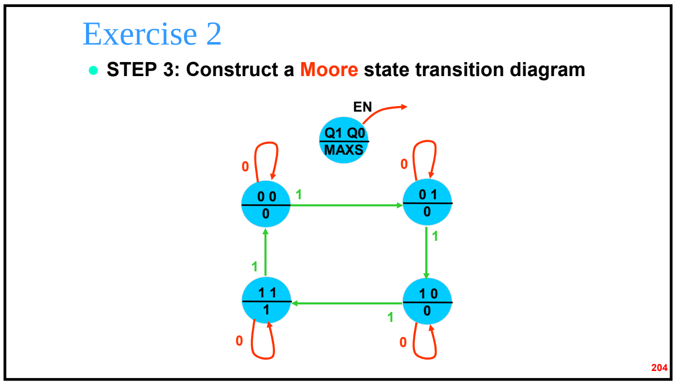
MAXS = Q0•Q1



Exercise 2

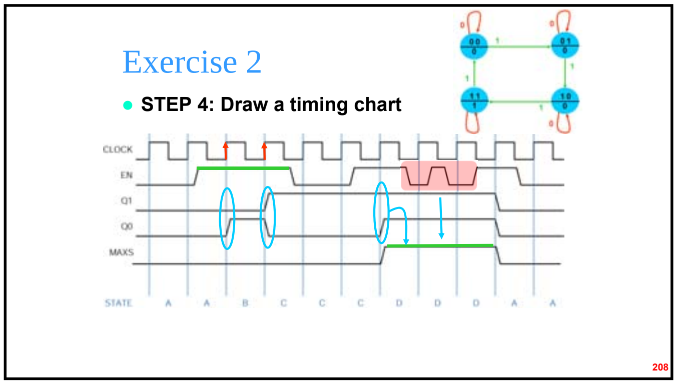
- STEP 2: Construct a PS-NS / O table

PS		PI	NS		Output
Q1	Q0	EN	Q1*	Q0*	MAXS
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1



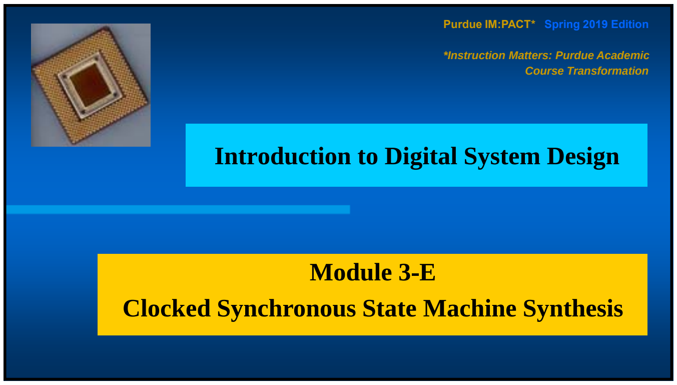
Exercise 2

- STEP 3: Construct a Moore state transition diagram



Exercise 2

- STEP 4: Draw a timing chart

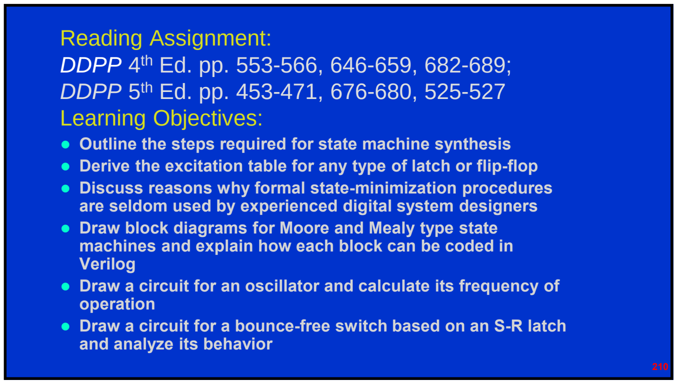


Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic
Course Transformation

Introduction to Digital System Design

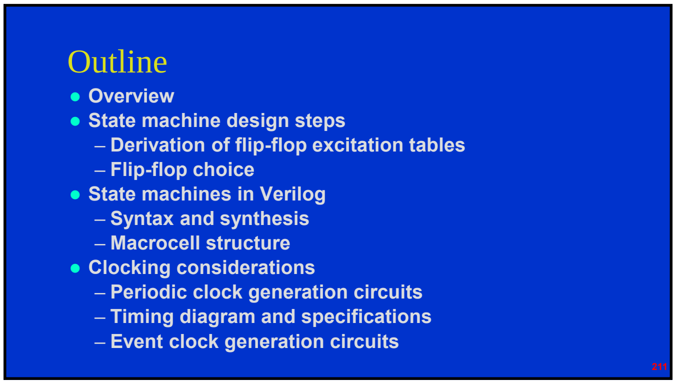
Module 3-E
Clocked Synchronous State Machine Synthesis



Reading Assignment:
DDPP 4th Ed. pp. 553-566, 646-659, 682-689;
DDPP 5th Ed. pp. 453-471, 676-680, 525-527

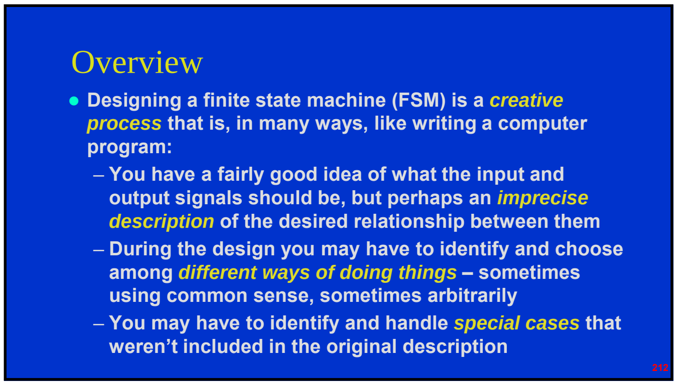
Learning Objectives:

- Outline the steps required for state machine synthesis
- Derive the excitation table for any type of latch or flip-flop
- Discuss reasons why formal state-minimization procedures are seldom used by experienced digital system designers
- Draw block diagrams for Moore and Mealy type state machines and explain how each block can be coded in Verilog
- Draw a circuit for an oscillator and calculate its frequency of operation
- Draw a circuit for a bounce-free switch based on an S-R latch and analyze its behavior



Outline

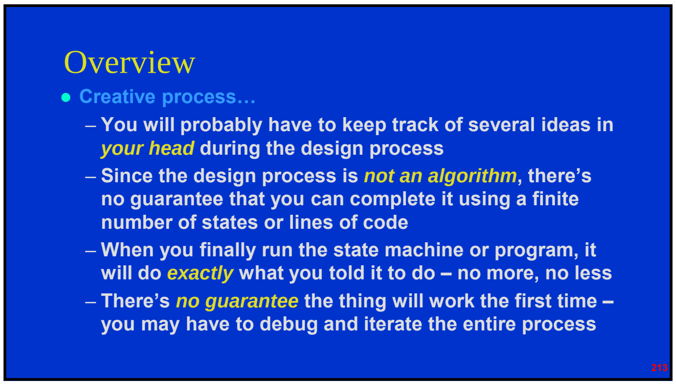
- Overview
- State machine design steps
  - Derivation of flip-flop excitation tables
  - Flip-flop choice
- State machines in Verilog
  - Syntax and synthesis
  - Macrocell structure
- Clocking considerations
  - Periodic clock generation circuits
  - Timing diagram and specifications
  - Event clock generation circuits



Overview

- Designing a finite state machine (FSM) is a *creative process* that is, in many ways, like writing a computer program:
  - You have a fairly good idea of what the input and output signals should be, but perhaps an *imprecise description* of the desired relationship between them
  - During the design you may have to identify and choose among *different ways of doing things* – sometimes using common sense, sometimes arbitrarily
  - You may have to identify and handle *special cases* that weren't included in the original description

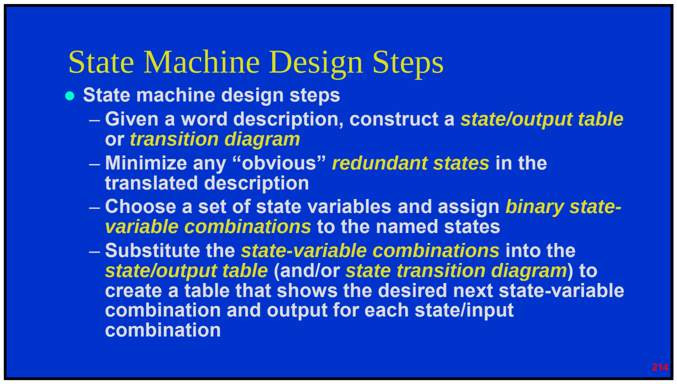
212



Overview

- Creative process…
  - You will probably have to keep track of several ideas in *your head* during the design process
  - Since the design process is *not an algorithm*, there's no guarantee that you can complete it using a finite number of states or lines of code
  - When you finally run the state machine or program, it will do *exactly* what you told it to do – no more, no less
  - There's *no guarantee* the thing will work the first time – you may have to debug and iterate the entire process

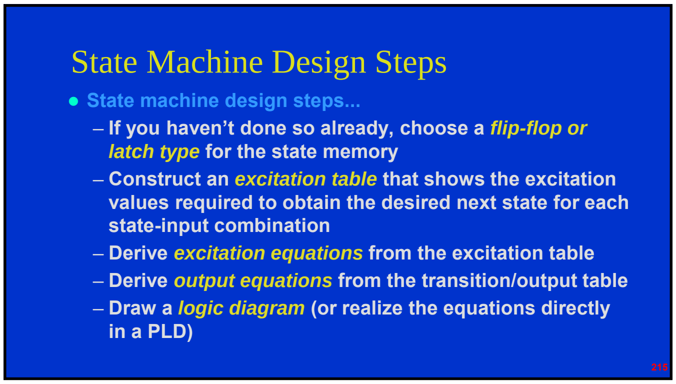
213



State Machine Design Steps

- State machine design steps
  - Given a word description, construct a *state/output table* or *transition diagram*
  - Minimize any "obvious" *redundant states* in the translated description
  - Choose a set of state variables and assign *binary state-variable combinations* to the named states
  - Substitute the *state-variable combinations* into the *state/output table* (and/or *state transition diagram*) to create a table that shows the desired next state-variable combination and output for each state/input combination

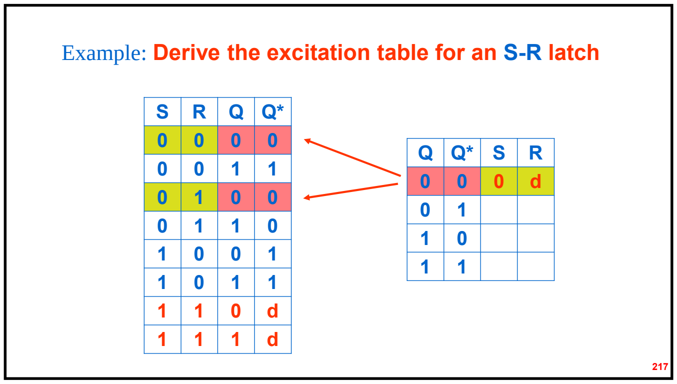
214



State Machine Design Steps

- State machine design steps...
  - If you haven't done so already, choose a *flip-flop or latch type* for the state memory
  - Construct an *excitation table* that shows the excitation values required to obtain the desired next state for each state-input combination
  - Derive *excitation equations* from the excitation table
  - Derive *output equations* from the transition/output table
  - Draw a *logic diagram* (or realize the equations directly in a PLD)

215

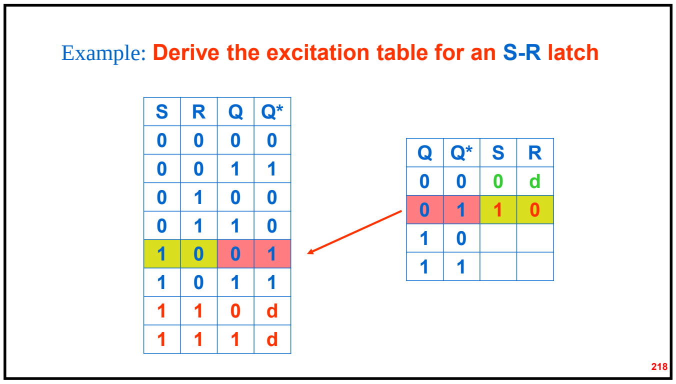


Example: **Derive the excitation table for an S-R latch**

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	d
1	1	1	d

Q	Q*	S	R
0	0	0	d
0	1		
1	0		
1	1		

217

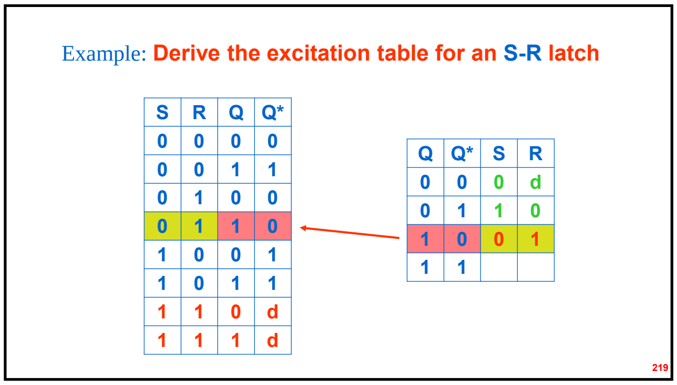


Example: **Derive the excitation table for an S-R latch**

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	d
1	1	1	d

Q	Q*	S	R
0	0	0	d
0	1	1	0
1	0		
1	1		

218

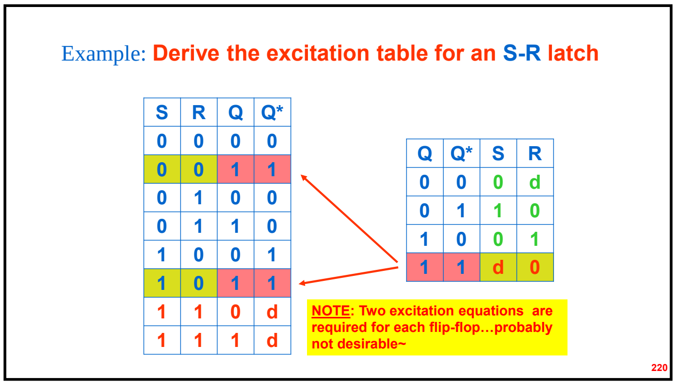


Example: **Derive the excitation table for an S-R latch**

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	d
1	1	1	d

Q	Q*	S	R
0	0	0	d
0	1	1	0
1	0	0	1
1	1		

219



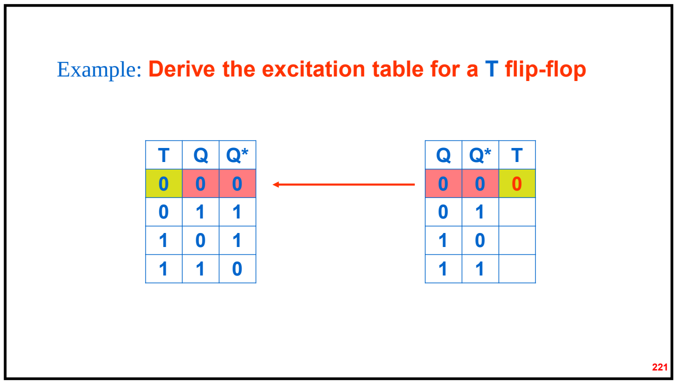
Example: **Derive the excitation table for an S-R latch**

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	d
1	1	1	d

Q	Q*	S	R
0	0	0	d
0	1	1	0
1	0	0	1
1	1	d	0

NOTE: Two excitation equations are required for each flip-flop…probably not desirable~

220

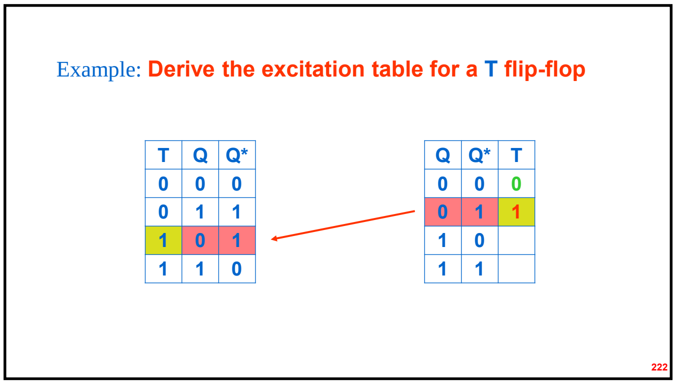


Example: **Derive the excitation table for a T flip-flop**

T	Q	Q*
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q*	T
0	0	0
0	1	
1	0	
1	1	

221

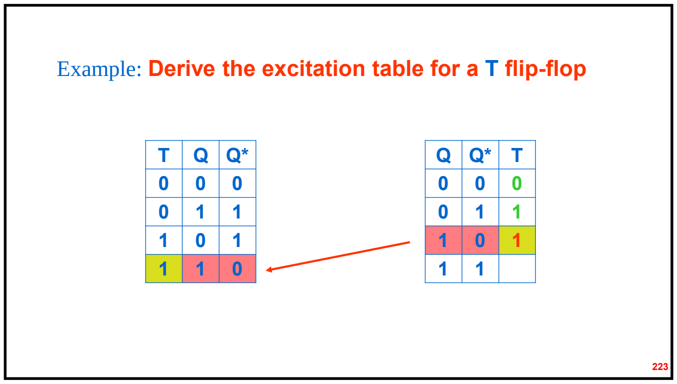


Example: **Derive the excitation table for a T flip-flop**

T	Q	Q*
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q*	T
0	0	0
0	1	1
1	0	
1	1	

222

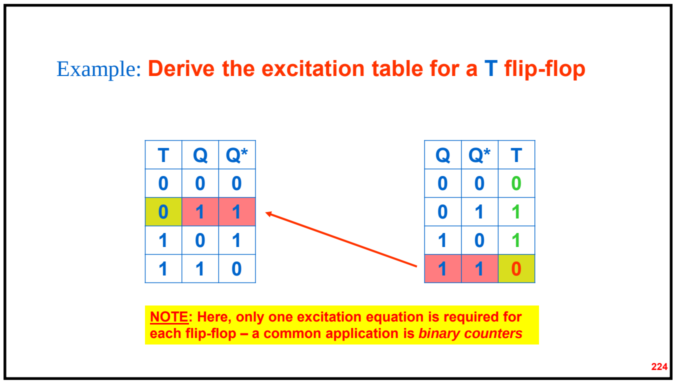


Example: **Derive the excitation table for a T flip-flop**

T	Q	Q*
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q*	T
0	0	0
0	1	1
1	0	1
1	1	

223



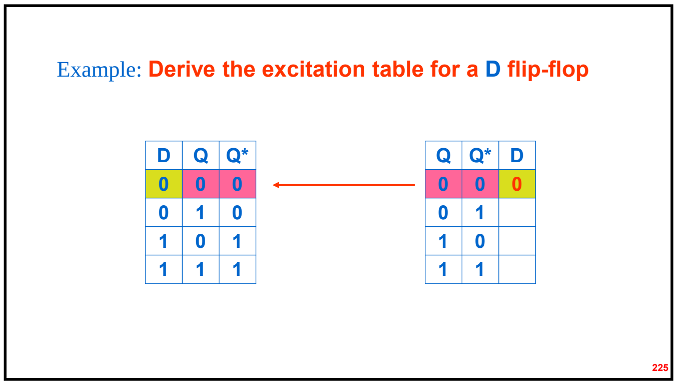
Example: **Derive the excitation table for a T flip-flop**

T	Q	Q*
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q*	T
0	0	0
0	1	1
1	0	1
1	1	0

NOTE: Here, only one excitation equation is required for each flip-flop – a common application is *binary counters*

224

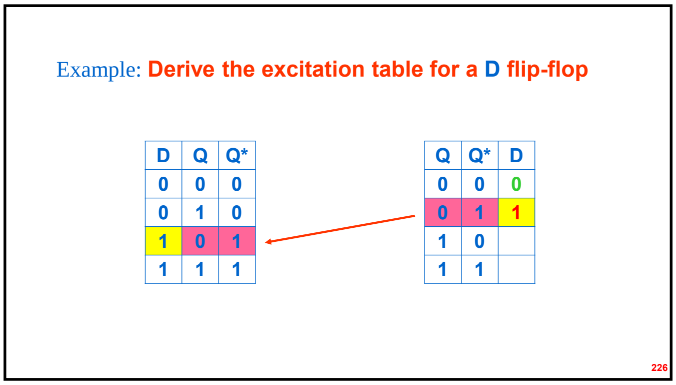


Example: **Derive the excitation table for a D flip-flop**

D	Q	Q*
0	0	0
0	1	0
1	0	1
1	1	1

Q	Q*	D
0	0	0
0	1	
1	0	
1	1	

225

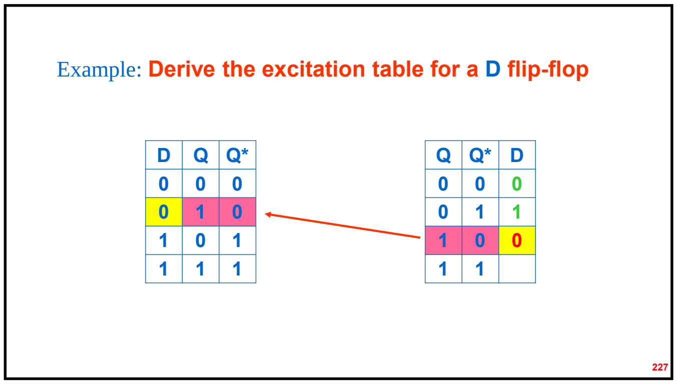


Example: **Derive the excitation table for a D flip-flop**

D	Q	Q*
0	0	0
0	1	0
1	0	1
1	1	1

Q	Q*	D
0	0	0
0	1	1
1	0	
1	1	

226

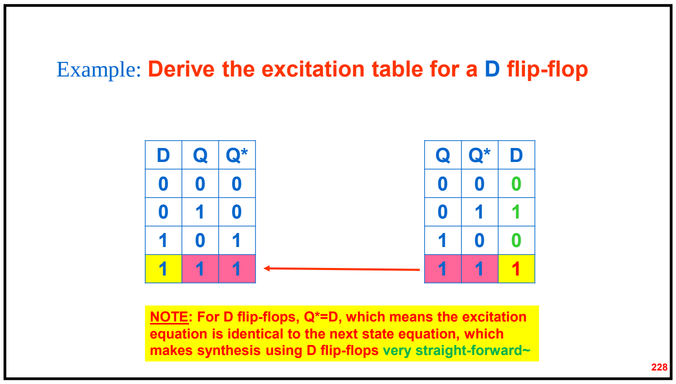


Example: **Derive the excitation table for a D flip-flop**

D	Q	Q*
0	0	0
0	1	0
1	0	1
1	1	1

Q	Q*	D
0	0	0
0	1	1
1	0	0
1	1	

227



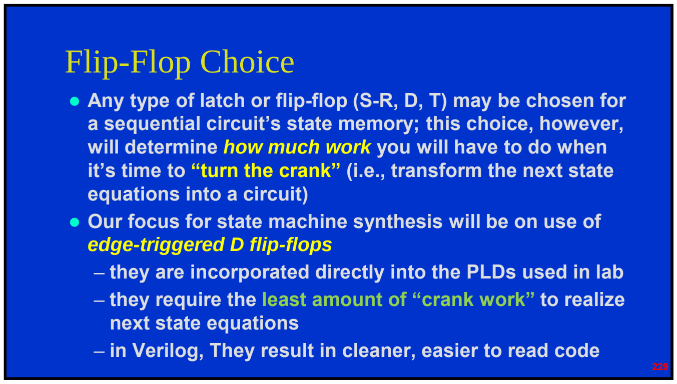
Example: **Derive the excitation table for a D flip-flop**

D	Q	Q*
0	0	0
0	1	0
1	0	1
1	1	1

Q	Q*	D
0	0	0
0	1	1
1	0	0
1	1	1

NOTE: For D flip-flops, Q*=D, which means the excitation equation is identical to the next state equation, which makes synthesis using D flip-flops very straight-forward~

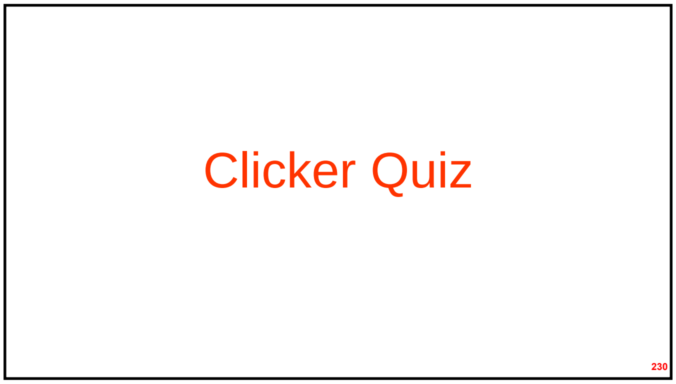
228



Flip-Flop Choice

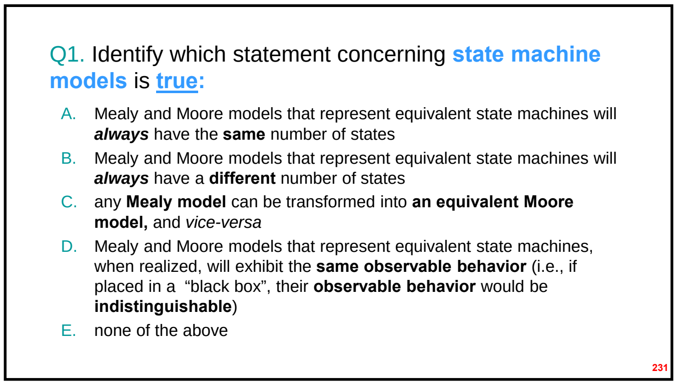
- Any type of latch or flip-flop (S-R, D, T) may be chosen for a sequential circuit's state memory; this choice, however, will determine ***how much work*** you will have to do when it's time to **"turn the crank"** (i.e., transform the next state equations into a circuit)
- Our focus for state machine synthesis will be on use of ***edge-triggered D flip-flops***
  - they are incorporated directly into the PLDs used in lab
  - they require the least amount of "crank work" to realize next state equations
  - in Verilog, They result in cleaner, easier to read code

229



Clicker Quiz

230



Q1. Identify which statement concerning **state machine models** is **true:**

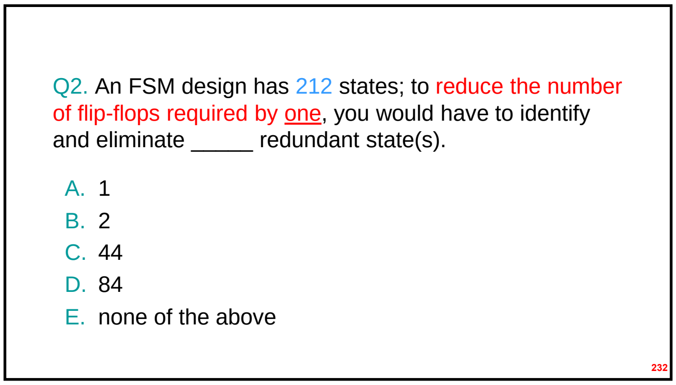
A. Mealy and Moore models that represent equivalent state machines will ***always*** have the **same** number of states

B. Mealy and Moore models that represent equivalent state machines will ***always*** have a **different** number of states

C. any **Mealy model** can be transformed into **an equivalent Moore model,** and *vice-versa*

D. Mealy and Moore models that represent equivalent state machines, when realized, will exhibit the **same observable behavior** (i.e., if placed in a "black box", their **observable behavior** would be **indistinguishable**)

E. none of the above



Q2. An FSM design has 212 states; to reduce the number of flip-flops required by one, you would have to identify and eliminate _____ redundant state(s).

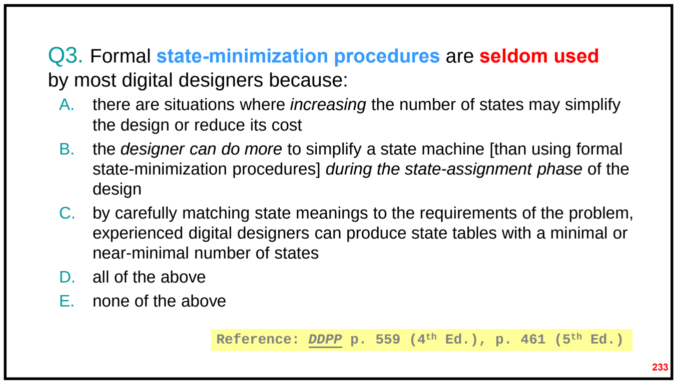
A. 1

B. 2

C. 44

D. 84

E. none of the above



Q3. Formal **state-minimization procedures** are **seldom used** by most digital designers because:

A. there are situations where *increasing* the number of states may simplify the design or reduce its cost

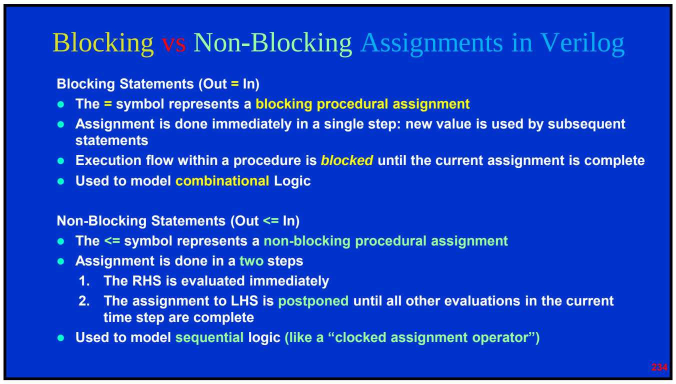
B. the *designer can do more* to simplify a state machine [than using formal state-minimization procedures] *during the state-assignment phase* of the design

C. by carefully matching state meanings to the requirements of the problem, experienced digital designers can produce state tables with a minimal or near-minimal number of states

D. all of the above

E. none of the above

Reference: ***DDPP*** p. 559 (4th Ed.), p. 461 (5th Ed.)



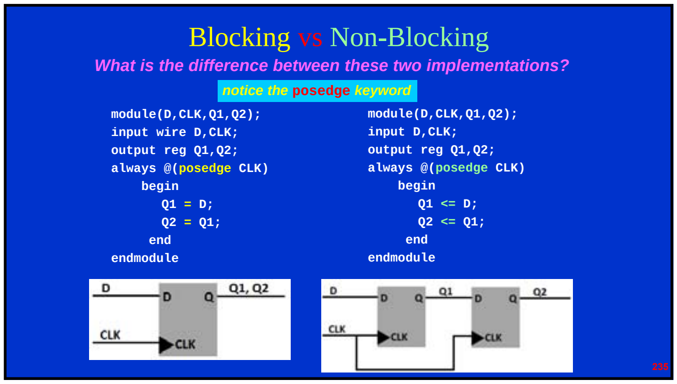
Blocking vs Non-Blocking Assignments in Verilog

Blocking Statements (Out = In)

- **The = symbol represents a blocking procedural assignment**
- **Assignment is done immediately in a single step: new value is used by subsequent statements**
- **Execution flow within a procedure is *blocked* until the current assignment is complete**
- **Used to model combinational Logic**

Non-Blocking Statements (Out <= In)

- **The <= symbol represents a non-blocking procedural assignment**
- **Assignment is done in a two steps**
  1. **The RHS is evaluated immediately**
  2. **The assignment to LHS is postponed until all other evaluations in the current time step are complete**
- **Used to model sequential logic (like a "clocked assignment operator")**



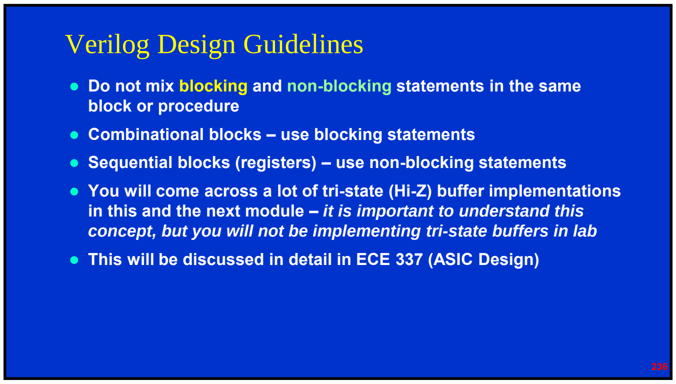
Blocking vs Non-Blocking

What is the difference between these two implementations?

notice the posedge keyword

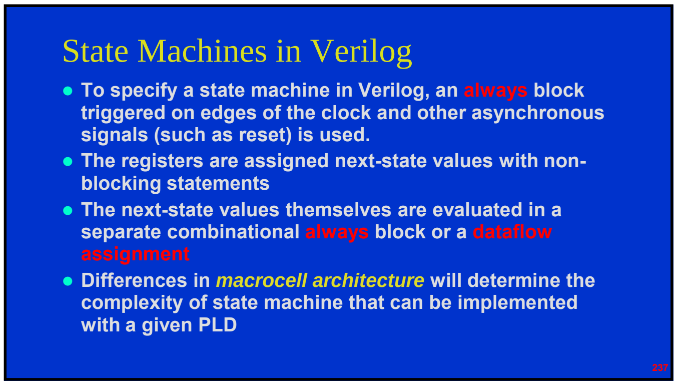
```
module(D,CLK,Q1,Q2);
input wire D,CLK;
output reg Q1,Q2;
always @(posedge CLK)
    begin
       Q1 = D;
       Q2 = Q1;
     end
endmodule
```

```
module(D,CLK,Q1,Q2);
input D,CLK;
output reg Q1,Q2;
always @(posedge CLK)
    begin
       Q1 <= D;
       Q2 <= Q1;
     end
endmodule
```



Verilog Design Guidelines

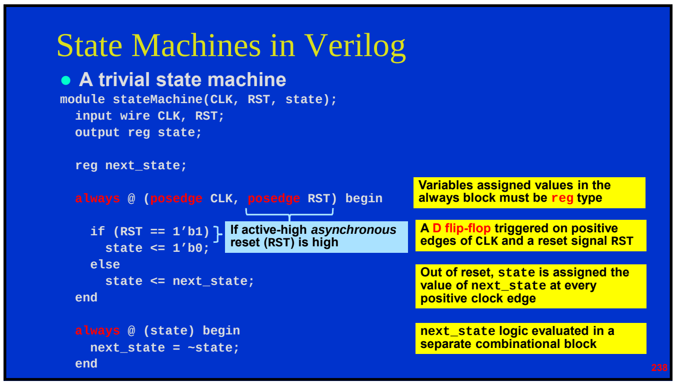
- **Do not mix blocking and non-blocking statements in the same block or procedure**
- **Combinational blocks – use blocking statements**
- **Sequential blocks (registers) – use non-blocking statements**
- **You will come across a lot of tri-state (Hi-Z) buffer implementations in this and the next module – *it is important to understand this concept, but you will not be implementing tri-state buffers in lab***
- **This will be discussed in detail in ECE 337 (ASIC Design)**



State Machines in Verilog

- To specify a state machine in Verilog, an always block triggered on edges of the clock and other asynchronous signals (such as reset) is used.
- The registers are assigned next-state values with non-blocking statements
- The next-state values themselves are evaluated in a separate combinational always block or a dataflow assignment
- Differences in *macrocell architecture* will determine the complexity of state machine that can be implemented with a given PLD

237



State Machines in Verilog

- A trivial state machine

```
module stateMachine(CLK, RST, state);
  input wire CLK, RST;
  output reg state;

  reg next_state;

  always @ (posedge CLK, posedge RST) begin

    if (RST == 1’b1)
      state <= 1’b0;
    else
      state <= next_state;
  end

  always @ (state) begin
    next_state = ~state;
  end
```

If active-high *asynchronous* reset (RST) is high

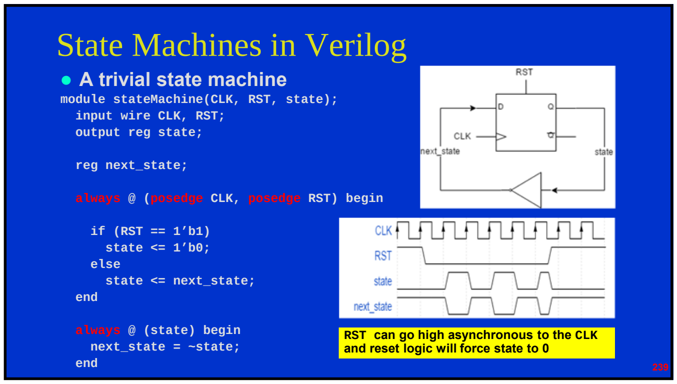
Variables assigned values in the always block must be reg type

A D flip-flop triggered on positive edges of CLK and a reset signal RST

Out of reset, state is assigned the value of next_state at every positive clock edge

next_state logic evaluated in a separate combinational block

238



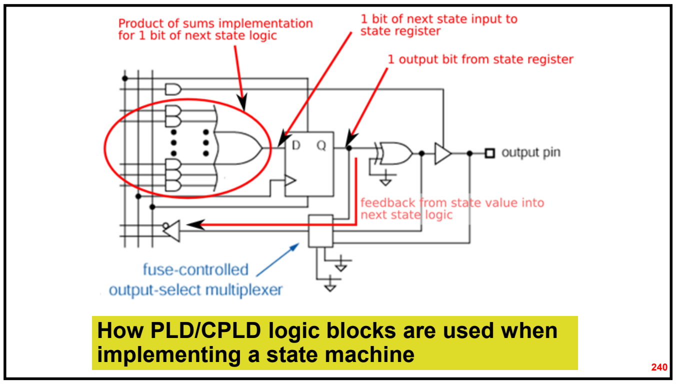
State Machines in Verilog

- A trivial state machine

```
module stateMachine(CLK, RST, state);
  input wire CLK, RST;
  output reg state;

  reg next_state;

  always @ (posedge CLK, posedge RST) begin

    if (RST == 1’b1)
      state <= 1’b0;
    else
      state <= next_state;
  end

  always @ (state) begin
    next_state = ~state;
  end
```

RST can go high asynchronous to the CLK and reset logic will force state to 0

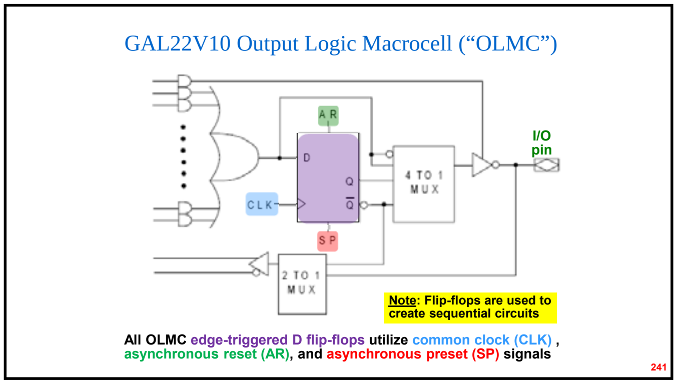
239



Product of sums implementation for 1 bit of next state logic
1 bit of next state input to state register
1 output bit from state register
D
Q
output pin
feedback from state value into next state logic
fuse-controlled output-select multiplexer

How PLD/CPLD logic blocks are used when implementing a state machine

240

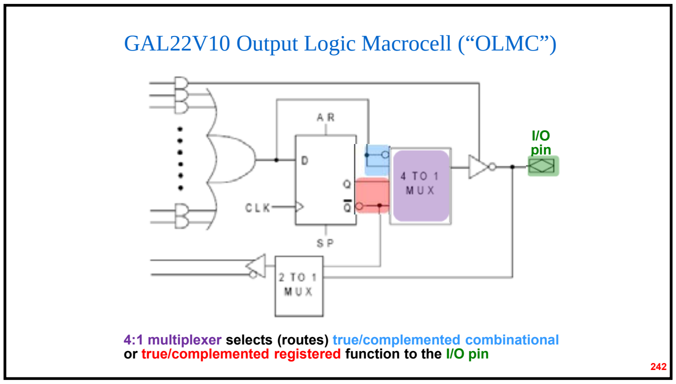


GAL22V10 Output Logic Macrocell (“OLMC”)

Note: Flip-flops are used to create sequential circuits

All OLMC edge-triggered D flip-flops utilize common clock (CLK) , asynchronous reset (AR), and asynchronous preset (SP) signals

241

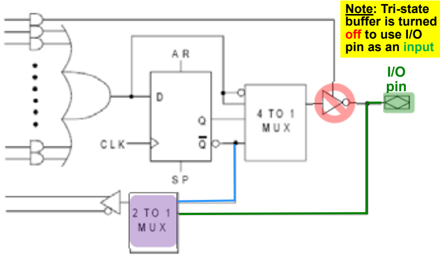


GAL22V10 Output Logic Macrocell (“OLMC”)

4:1 multiplexer selects (routes) true/complemented combinational or true/complemented registered function to the I/O pin

242

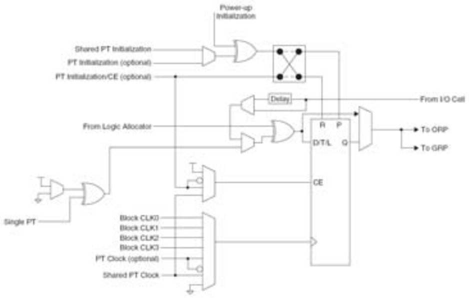
GAL22V10 Output Logic Macrocell ("OLMC")



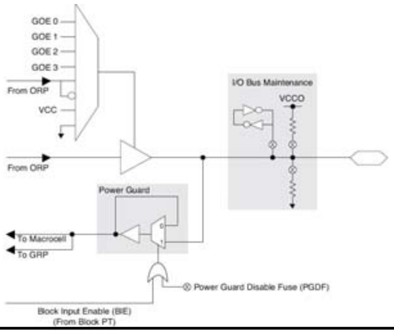
Note: Tri-state buffer is turned off to use I/O pin as an input

2:1 multiplexer selects (routes) true/complemented I/O pin or true/complemented registered feedback to the P-term array

ispMACH 4000ZE Macrocell



ispMACH 4000ZE I/O Cell



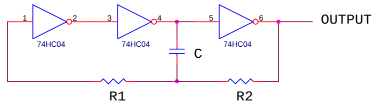
Clocking Considerations

- State machines require a *clocking signal* in order to operate "sequentially"
- There are two basic types of clocking signals that can be used:
  - *periodic* ("continuously running"), generated using an *oscillator circuit*
  - *event* (non-periodic, single clock edge), generated using a *bounce-free* switch or sensor contact closure
- A *timing diagram* can be used to show the relationship between the clock and various input, output, and internal signals – it can also be used to help answer the key question facing computer system designers: *"How fast can this thing run?"*

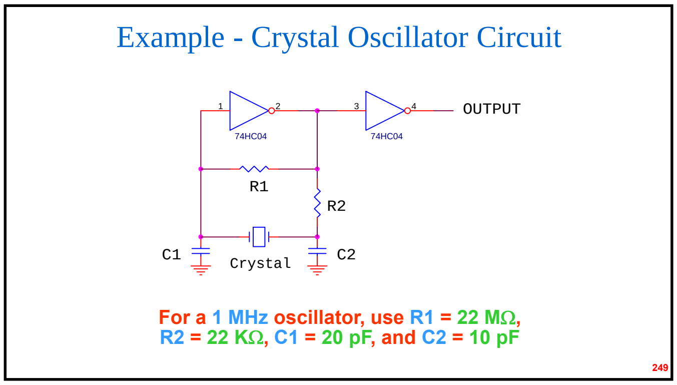
Periodic Clock Generation Circuits

- Periodic clock signals can be generated using several different types of *oscillator* circuits:
  - based on an R-C time constant *(least accurate)*
  - based on a ceramic resonator
  - based on a quartz crystal *(most accurate)*
- Issues of interest include the following:
  - frequency of operation
  - duty cycle
  - transition time
  - ringing (undershoot / overshoot)
  - stability (long term drift / short term "jitter")
  - driving capability / need for buffers
  - skew (different length paths on PCB)

Example - CMOS "Ring" Oscillator



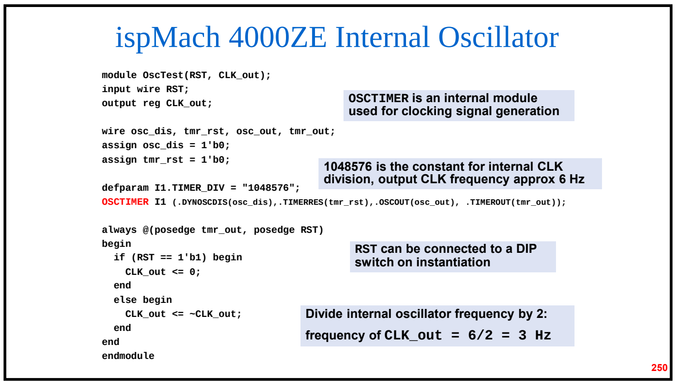
$f \cong (2C(0.4R_{eq} + 0.7R_2))^{-1}$ where $R_{eq} = (R_1R_2)/(R_1+R_2)$



Example - Crystal Oscillator Circuit

For a 1 MHz oscillator, use R1 = 22 MΩ, R2 = 22 KΩ, C1 = 20 pF, and C2 = 10 pF

249



ispMach 4000ZE Internal Oscillator

```
module OscTest(RST, CLK_out);
input wire RST;
output reg CLK_out;

wire osc_dis, tmr_rst, osc_out, tmr_out;
assign osc_dis = 1'b0;
assign tmr_rst = 1'b0;

defparam I1.TIMER_DIV = "1048576";
OSCTIMER I1 (.DYNOSCDIS(osc_dis),.TIMERRES(tmr_rst),.OSCOUT(osc_out), .TIMEROUT(tmr_out));

always @(posedge tmr_out, posedge RST)
begin
  if (RST == 1'b1) begin
    CLK_out <= 0;
  end
  else begin
    CLK_out <= ~CLK_out;
  end
end
endmodule
```

OSCTIMER is an internal module used for clocking signal generation

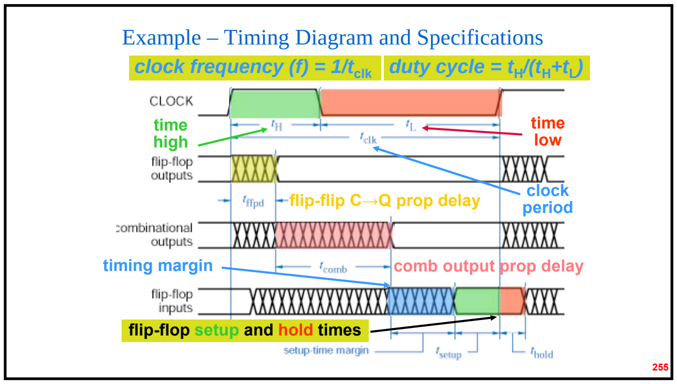
1048576 is the constant for internal CLK division, output CLK frequency approx 6 Hz

RST can be connected to a DIP switch on instantiation

Divide internal oscillator frequency by 2:

frequency of CLK_out = 6/2 = 3 Hz

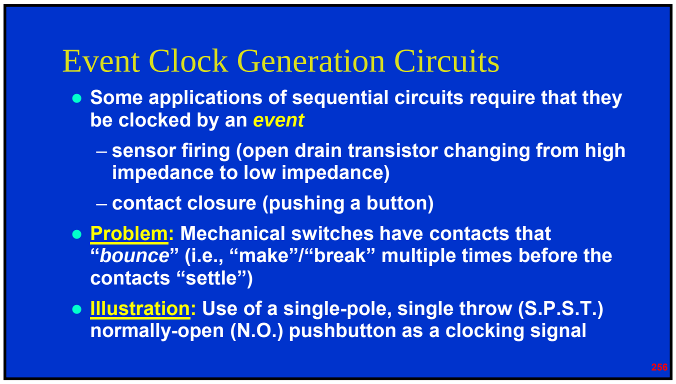
250



Example – Timing Diagram and Specifications

clock frequency (f) = 1/t_{clk}   *duty cycle = $t_H/(t_H+t_L)$*

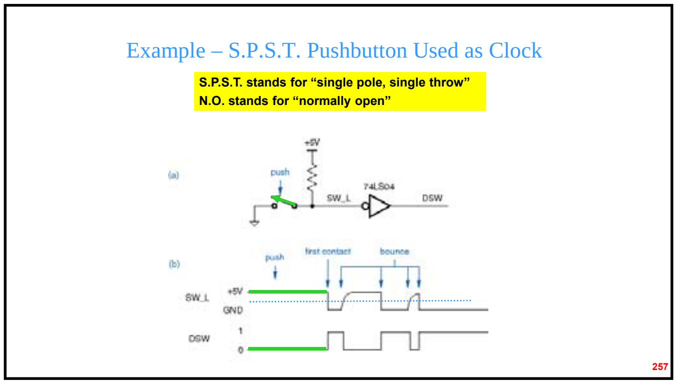
255



Event Clock Generation Circuits

- Some applications of sequential circuits require that they be clocked by an ***event***
  - sensor firing (open drain transistor changing from high impedance to low impedance)
  - contact closure (pushing a button)
- Problem: Mechanical switches have contacts that "***bounce***" (i.e., "make"/"break" multiple times before the contacts "settle")
- Illustration: Use of a single-pole, single throw (S.P.S.T.) normally-open (N.O.) pushbutton as a clocking signal

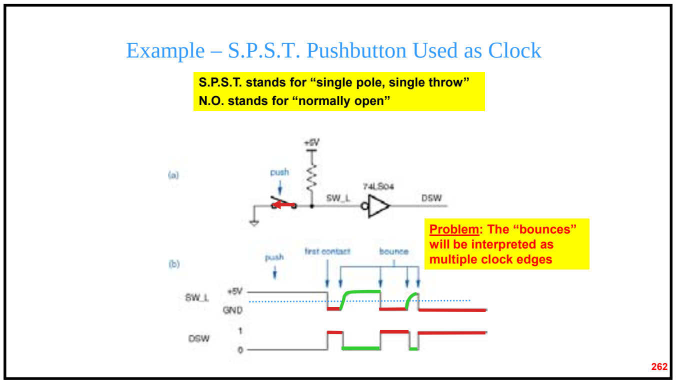
256



Example – S.P.S.T. Pushbutton Used as Clock

S.P.S.T. stands for "single pole, single throw"
N.O. stands for "normally open"

257

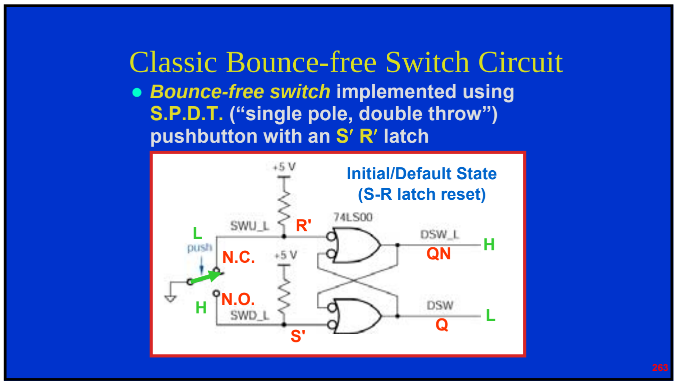


Example – S.P.S.T. Pushbutton Used as Clock

S.P.S.T. stands for "single pole, single throw"
N.O. stands for "normally open"

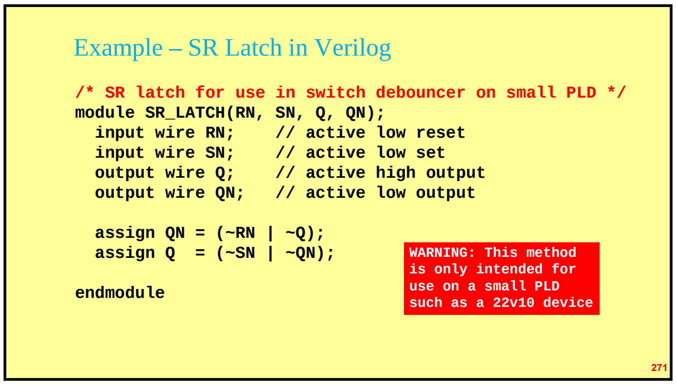
Problem: The "bounces" will be interpreted as multiple clock edges

262



Classic Bounce-free Switch Circuit

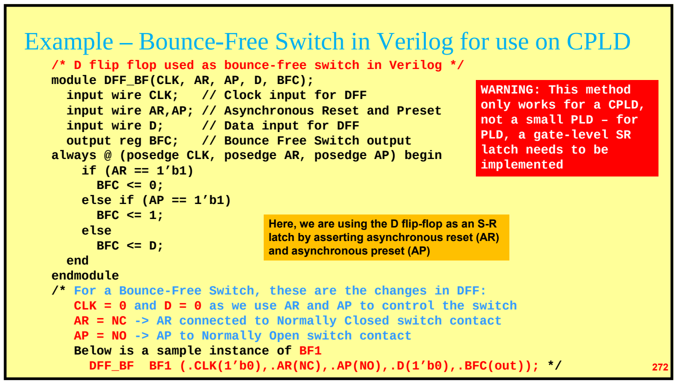
- ***Bounce-free switch*** **implemented using S.P.D.T. (“single pole, double throw”) pushbutton with an S′ R′ latch**



Example – SR Latch in Verilog

```
/* SR latch for use in switch debouncer on small PLD */
module SR_LATCH(RN, SN, Q, QN);
  input wire RN;    // active low reset
  input wire SN;    // active low set
  output wire Q;    // active high output
  output wire QN;   // active low output

  assign QN = (~RN | ~Q);
  assign Q  = (~SN | ~QN);

endmodule
```

WARNING: This method is only intended for use on a small PLD such as a 22v10 device

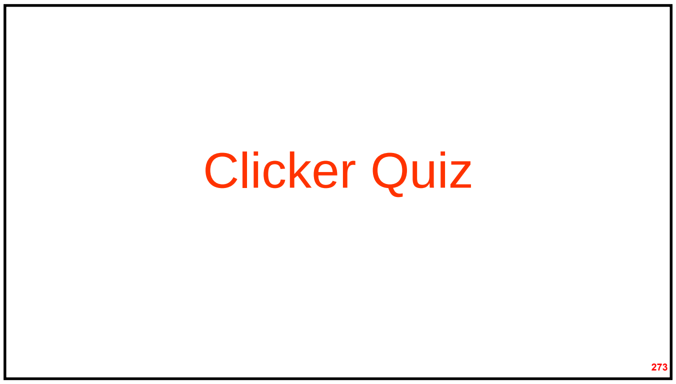


Example – Bounce-Free Switch in Verilog for use on CPLD

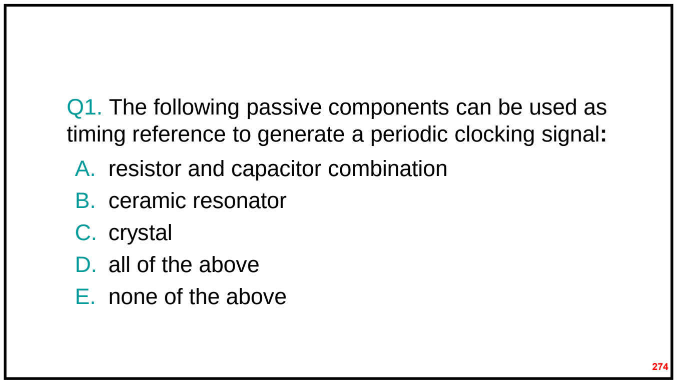
```
/* D flip flop used as bounce-free switch in Verilog */
module DFF_BF(CLK, AR, AP, D, BFC);
  input wire CLK;   // Clock input for DFF
  input wire AR,AP; // Asynchronous Reset and Preset
  input wire D;     // Data input for DFF
  output reg BFC;   // Bounce Free Switch output
always @ (posedge CLK, posedge AR, posedge AP) begin
    if (AR == 1’b1)
      BFC <= 0;
    else if (AP == 1’b1)
      BFC <= 1;
    else
      BFC <= D;
  end
endmodule
/* For a Bounce-Free Switch, these are the changes in DFF:
   CLK = 0 and D = 0 as we use AR and AP to control the switch
   AR = NC -> AR connected to Normally Closed switch contact
   AP = NO -> AP to Normally Open switch contact
   Below is a sample instance of BF1
     DFF_BF  BF1 (.CLK(1’b0),.AR(NC),.AP(NO),.D(1’b0),.BFC(out)); */
```

WARNING: This method only works for a CPLD, not a small PLD – for PLD, a gate-level SR latch needs to be implemented

Here, we are using the D flip-flop as an S-R latch by asserting asynchronous reset (AR) and asynchronous preset (AP)

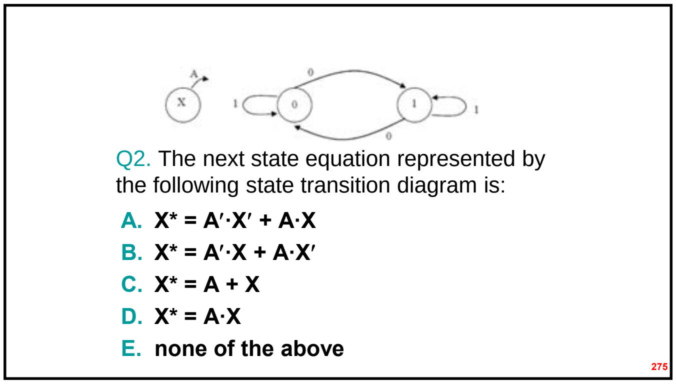


Clicker Quiz



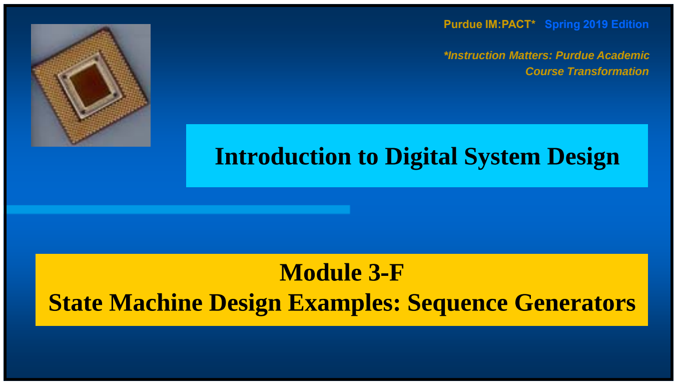
Q1. The following passive components can be used as timing reference to generate a periodic clocking signal**:**

A. resistor and capacitor combination
B. ceramic resonator
C. crystal
D. all of the above
E. none of the above



Q2. The next state equation represented by the following state transition diagram is:

A. X* = A′·X′ + A·X
B. X* = A′·X + A·X′
C. X* = A + X
D. X* = A·X
E. none of the above

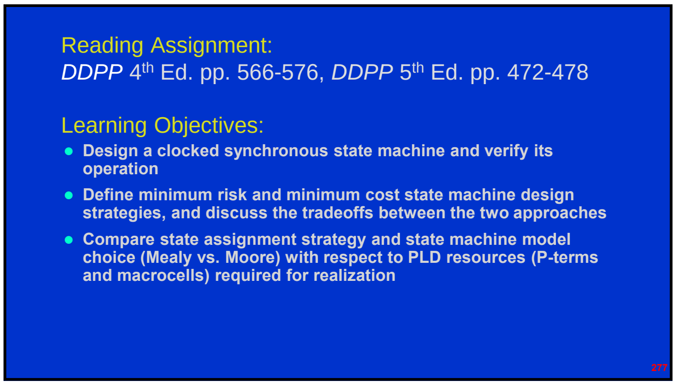


Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 3-F
State Machine Design Examples: Sequence Generators

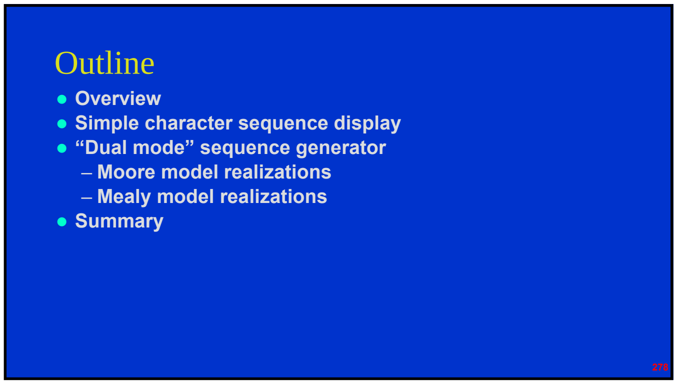


Reading Assignment:
DDPP 4th Ed. pp. 566-576, *DDPP* 5th Ed. pp. 472-478

Learning Objectives:

- Design a clocked synchronous state machine and verify its operation
- Define minimum risk and minimum cost state machine design strategies, and discuss the tradeoffs between the two approaches
- Compare state assignment strategy and state machine model choice (Mealy vs. Moore) with respect to PLD resources (P-terms and macrocells) required for realization

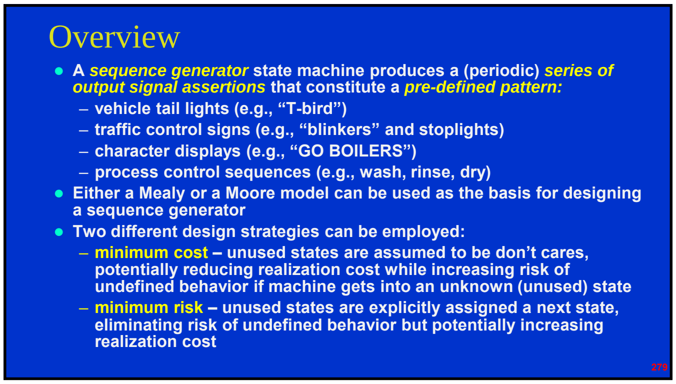
277



Outline

- Overview
- Simple character sequence display
- “Dual mode” sequence generator
  - Moore model realizations
  - Mealy model realizations
- Summary

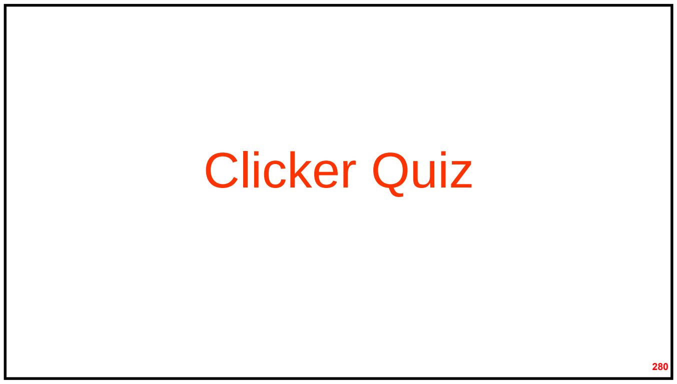
278



Overview

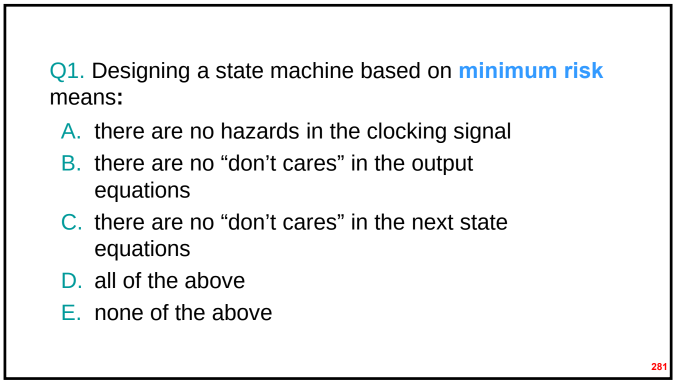
- A ***sequence generator*** state machine produces a (periodic) ***series of output signal assertions*** that constitute a ***pre-defined pattern:***
  - vehicle tail lights (e.g., “T-bird”)
  - traffic control signs (e.g., “blinkers” and stoplights)
  - character displays (e.g., “GO BOILERS”)
  - process control sequences (e.g., wash, rinse, dry)
- Either a Mealy or a Moore model can be used as the basis for designing a sequence generator
- Two different design strategies can be employed:
  - **minimum cost** – unused states are assumed to be don’t cares, potentially reducing realization cost while increasing risk of undefined behavior if machine gets into an unknown (unused) state
  - **minimum risk** – unused states are explicitly assigned a next state, eliminating risk of undefined behavior but potentially increasing realization cost

279



Clicker Quiz

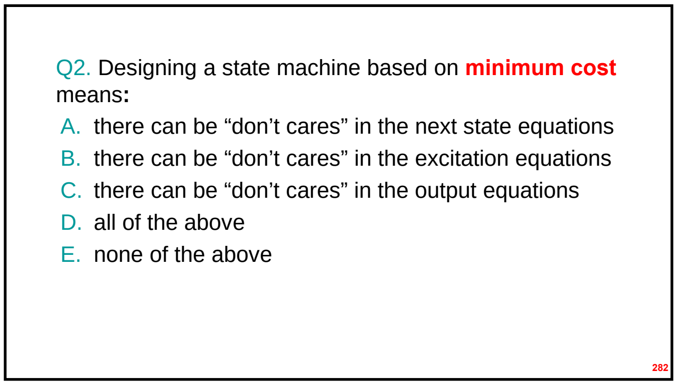
280



Q1. Designing a state machine based on **minimum risk** means**:**

A. there are no hazards in the clocking signal
B. there are no “don’t cares” in the output equations
C. there are no “don’t cares” in the next state equations
D. all of the above
E. none of the above

281



Q2. Designing a state machine based on **minimum cost** means**:**

A. there can be “don’t cares” in the next state equations

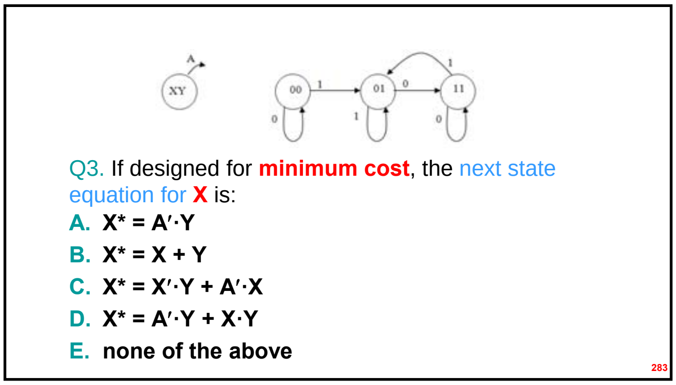
B. there can be “don’t cares” in the excitation equations

C. there can be “don’t cares” in the output equations

D. all of the above

E. none of the above

282



Q3. If designed for **minimum cost**, the next state equation for **X** is:

A. X* = A′·Y

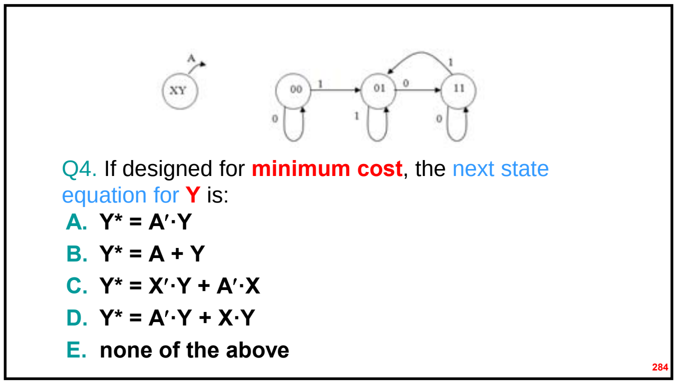
B. X* = X + Y

C. X* = X′·Y + A′·X

D. X* = A′·Y + X·Y

E. none of the above

283



Q4. If designed for **minimum cost**, the next state equation for **Y** is:

A. Y* = A′·Y

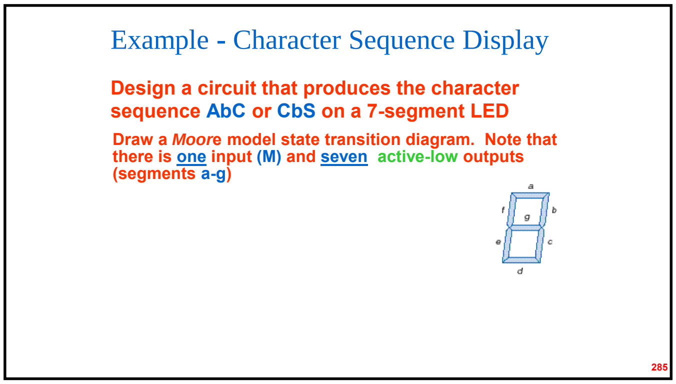
B. Y* = A + Y

C. Y* = X′·Y + A′·X

D. Y* = A′·Y + X·Y

E. none of the above

284

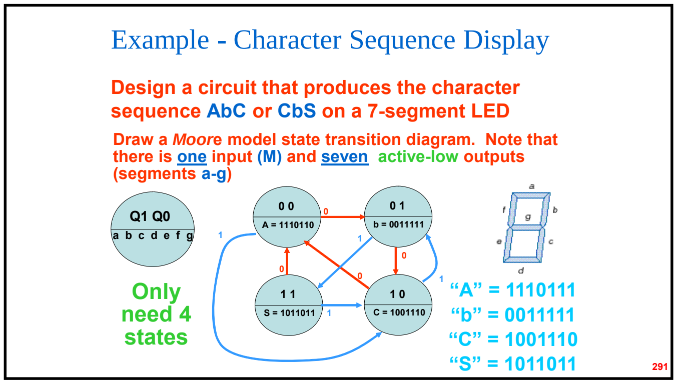


Example - Character Sequence Display

Design a circuit that produces the character sequence AbC or CbS on a 7-segment LED

Draw a *Moore* model state transition diagram. Note that there is <u>one</u> input (M) and <u>seven</u> active-low outputs (segments a-g)

285



Example - Character Sequence Display

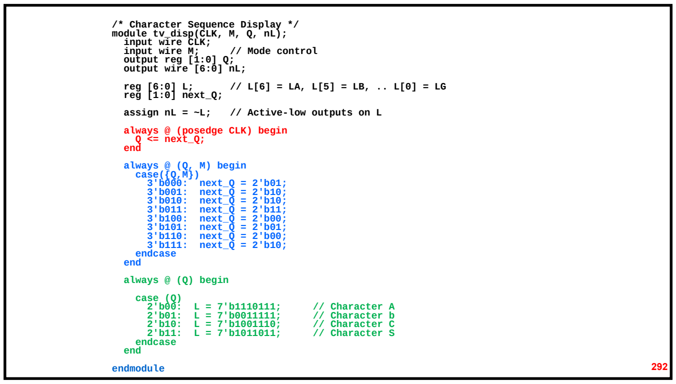
Design a circuit that produces the character sequence AbC or CbS on a 7-segment LED

Draw a *Moore* model state transition diagram. Note that there is <u>one</u> input (M) and <u>seven</u> active-low outputs (segments a-g)

Only need 4 states

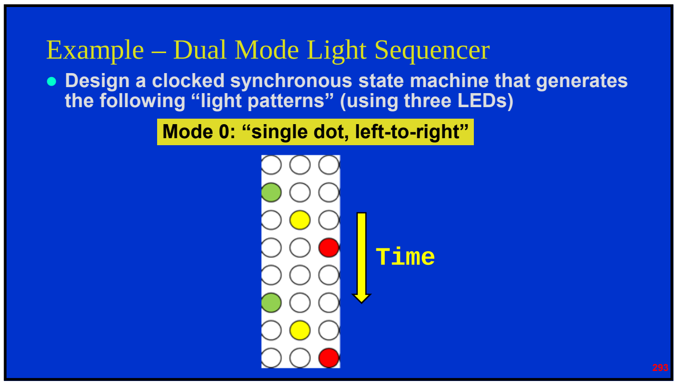
“A” = 1110111
“b” = 0011111
“C” = 1001110
“S” = 1011011

291



```
/* Character Sequence Display */
module tv_disp(CLK, M, Q, nL);
  input wire CLK;
  input wire M;     // Mode control
  output reg [1:0] Q;
  output wire [6:0] nL;

  reg [6:0] L;      // L[6] = LA, L[5] = LB, .. L[0] = LG
  reg [1:0] next_Q;

  assign nL = ~L;   // Active-low outputs on L

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q, M) begin
    case({Q,M})
      3'b000:  next_Q = 2'b01;
      3'b001:  next_Q = 2'b10;
      3'b010:  next_Q = 2'b10;
      3'b011:  next_Q = 2'b11;
      3'b100:  next_Q = 2'b00;
      3'b101:  next_Q = 2'b01;
      3'b110:  next_Q = 2'b00;
      3'b111:  next_Q = 2'b10;
    endcase
  end

  always @ (Q) begin

    case (Q)
      2'b00:  L = 7'b1110111;     // Character A
      2'b01:  L = 7'b0011111;     // Character b
      2'b10:  L = 7'b1001110;     // Character C
      2'b11:  L = 7'b1011011;     // Character S
    endcase
  end

endmodule
```

292

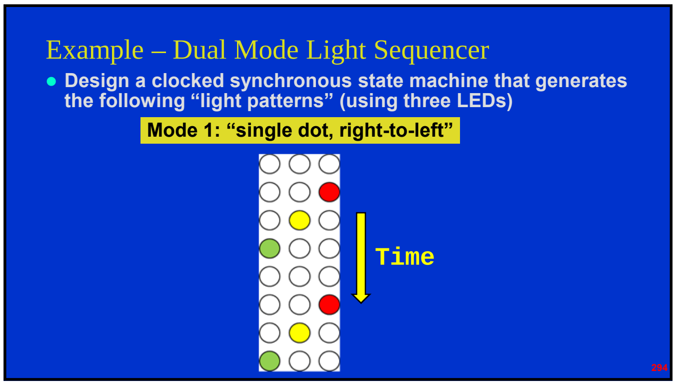


Example – Dual Mode Light Sequencer

- Design a clocked synchronous state machine that generates the following “light patterns” (using three LEDs)

Mode 0: “single dot, left-to-right”

293

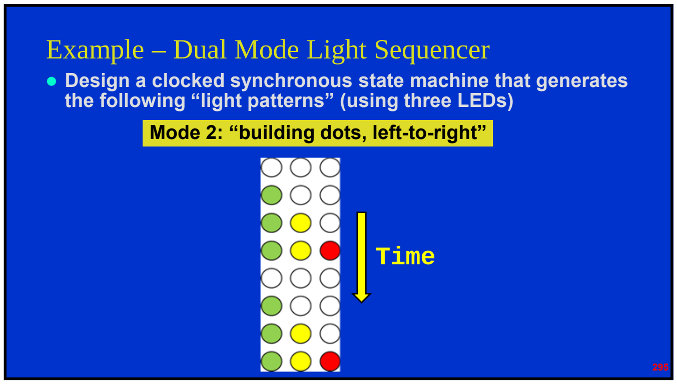


Example – Dual Mode Light Sequencer

- Design a clocked synchronous state machine that generates the following “light patterns” (using three LEDs)

Mode 1: “single dot, right-to-left”

294

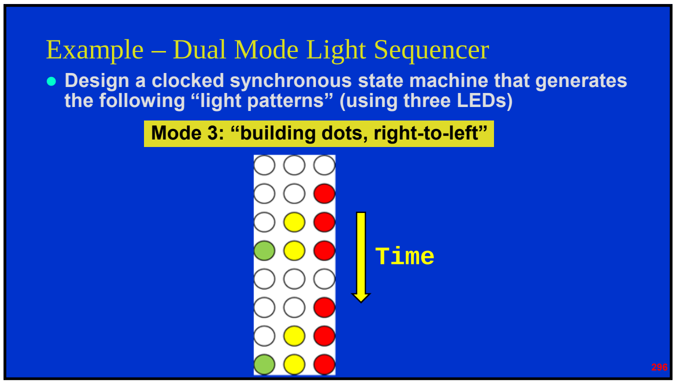


Example – Dual Mode Light Sequencer

- Design a clocked synchronous state machine that generates the following “light patterns” (using three LEDs)

Mode 2: “building dots, left-to-right”

295



Example – Dual Mode Light Sequencer

- Design a clocked synchronous state machine that generates the following “light patterns” (using three LEDs)

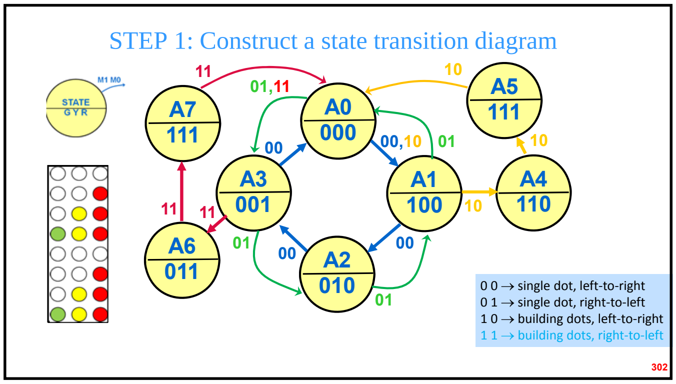
Mode 3: “building dots, right-to-left”

296

Moore Model Realizations

- To specify in which of the 4 modes we want the circuit to operate, we will need 2 “mode control” inputs, M1 and M0, where:
  - 0 0 → single dot, left-to-right
  - 0 1 → single dot, right-to-left
  - 1 0 → building dots, left-to-right
  - 1 1 → building dots, right-to-left
- A separate output function needs to be determined for each of the 3 LED outputs: G, Y, and R (from left-to-right)
- A state will be needed corresponding to the “all LEDs off” condition

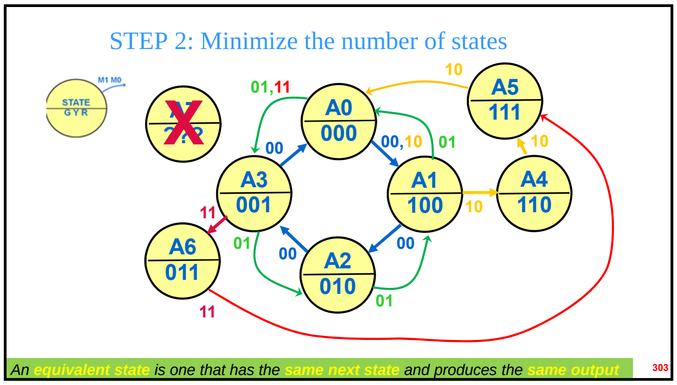
297



STEP 1: Construct a state transition diagram

0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left

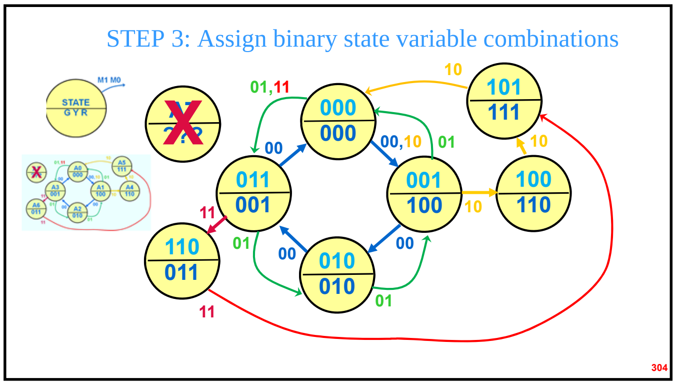
302



STEP 2: Minimize the number of states

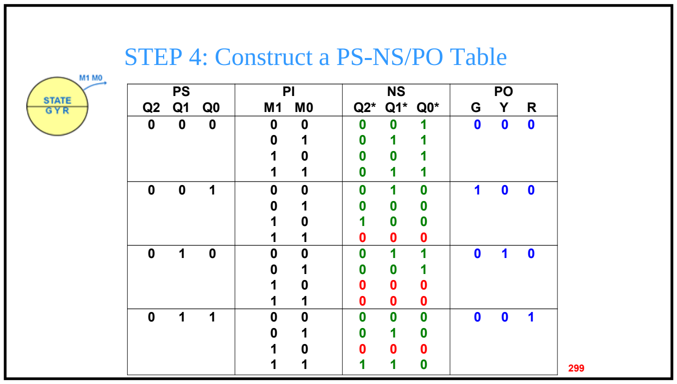
An equivalent state is one that has the same next state and produces the same output

303



STEP 3: Assign binary state variable combinations

304

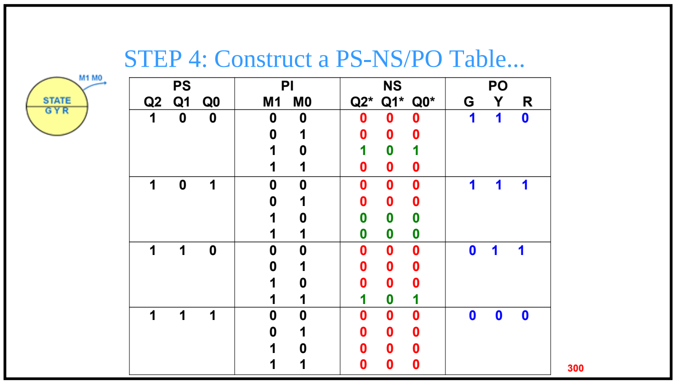


STEP 4: Construct a PS-NS/PO Table

M1 M0
STATE
G Y R

PS			PI		NS			PO		
Q2	Q1	Q0	M1	M0	Q2*	Q1*	Q0*	G	Y	R
0	0	0	0	0	0	0	1	0	0	0
			0	1	0	1	1			
			1	0	0	0	1			
			1	1	0	1	1			
0	0	1	0	0	0	1	0	1	0	0
			0	1	0	0	0			
			1	0	1	0	0			
			1	1	0	0	0			
0	1	0	0	0	0	1	1	0	1	0
			0	1	0	0	1			
			1	0	0	0	0			
			1	1	0	0	0			
0	1	1	0	0	0	0	0	0	0	1
			0	1	0	1	0			
			1	0	0	0	0			
			1	1	1	1	0			

299

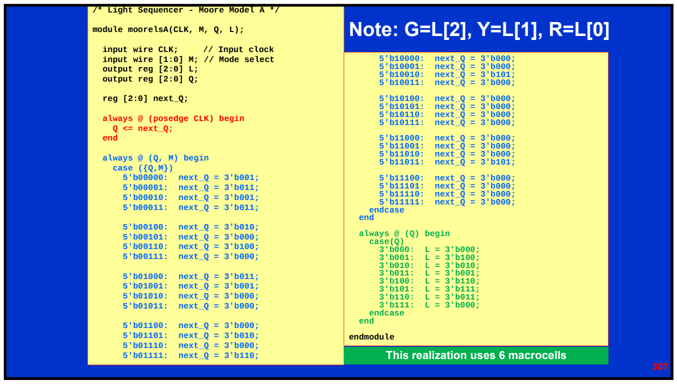


STEP 4: Construct a PS-NS/PO Table...

M1 M0
STATE
G Y R

PS			PI		NS			PO		
Q2	Q1	Q0	M1	M0	Q2*	Q1*	Q0*	G	Y	R
1	0	0	0	0	0	0	0	1	1	0
			0	1	0	0	0			
			1	0	1	0	1			
			1	1	0	0	0			
1	0	1	0	0	0	0	0	1	1	1
			0	1	0	0	0			
			1	0	0	0	0			
			1	1	0	0	0			
1	1	0	0	0	0	0	0	0	1	1
			0	1	0	0	0			
			1	0	0	0	0			
			1	1	1	0	1			
1	1	1	0	0	0	0	0	0	0	0
			0	1	0	0	0			
			1	0	0	0	0			
			1	1	0	0	0			

300



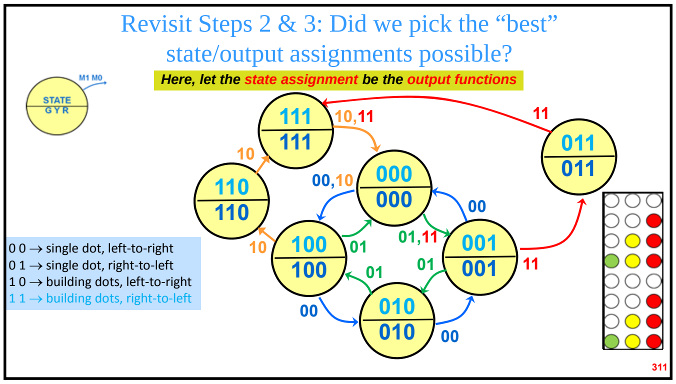
```
/* Light Sequencer - Moore Model A */

module moorelsA(CLK, M, Q, L);

  input wire CLK;     // Input clock
  input wire [1:0] M; // Mode select
  output reg [2:0] L;
  output reg [2:0] Q;

  reg [2:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q, M) begin
    case ({Q,M})
      5'b00000:  next_Q = 3'b001;
      5'b00001:  next_Q = 3'b011;
      5'b00010:  next_Q = 3'b001;
      5'b00011:  next_Q = 3'b011;

      5'b00100:  next_Q = 3'b010;
      5'b00101:  next_Q = 3'b000;
      5'b00110:  next_Q = 3'b100;
      5'b00111:  next_Q = 3'b000;

      5'b01000:  next_Q = 3'b011;
      5'b01001:  next_Q = 3'b001;
      5'b01010:  next_Q = 3'b000;
      5'b01011:  next_Q = 3'b000;

      5'b01100:  next_Q = 3'b000;
      5'b01101:  next_Q = 3'b010;
      5'b01110:  next_Q = 3'b000;
      5'b01111:  next_Q = 3'b110;
```

Note: G=L[2], Y=L[1], R=L[0]

```
      5'b10000:  next_Q = 3'b000;
      5'b10001:  next_Q = 3'b000;
      5'b10010:  next_Q = 3'b101;
      5'b10011:  next_Q = 3'b000;

      5'b10100:  next_Q = 3'b000;
      5'b10101:  next_Q = 3'b000;
      5'b10110:  next_Q = 3'b000;
      5'b10111:  next_Q = 3'b000;

      5'b11000:  next_Q = 3'b000;
      5'b11001:  next_Q = 3'b000;
      5'b11010:  next_Q = 3'b000;
      5'b11011:  next_Q = 3'b101;

      5'b11100:  next_Q = 3'b000;
      5'b11101:  next_Q = 3'b000;
      5'b11110:  next_Q = 3'b000;
      5'b11111:  next_Q = 3'b000;
    endcase
  end

  always @ (Q) begin
    case(Q)
      3'b000:  L = 3'b000;
      3'b001:  L = 3'b100;
      3'b010:  L = 3'b010;
      3'b011:  L = 3'b001;
      3'b100:  L = 3'b110;
      3'b101:  L = 3'b111;
      3'b110:  L = 3'b011;
      3'b111:  L = 3'b000;
    endcase
  end

endmodule
```

This realization uses 6 macrocells

307

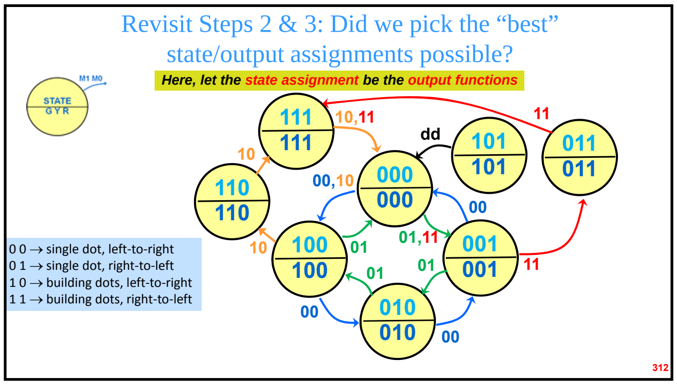


Revisit Steps 2 & 3: Did we pick the "best" state/output assignments possible?

Here, let the state assignment be the output functions

0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left

311

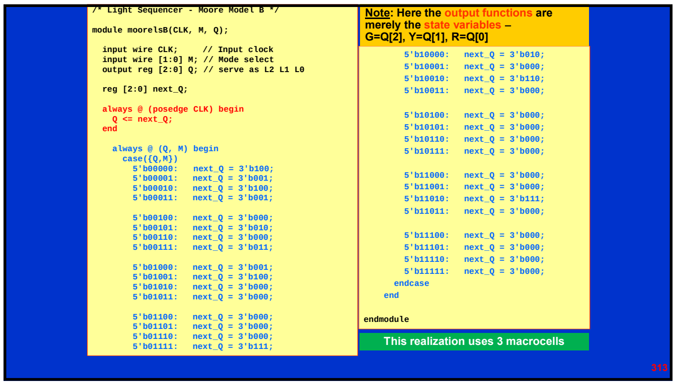


Revisit Steps 2 & 3: Did we pick the “best” state/output assignments possible?

Here, let the state assignment be the output functions

0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left

312



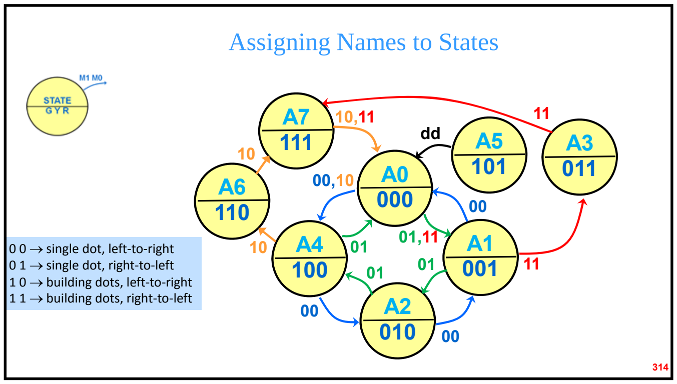
```
/* Light Sequencer - Moore Model B */

module moorelsB(CLK, M, Q);

  input wire CLK;     // Input clock
  input wire [1:0] M; // Mode select
  output reg [2:0] Q; // serve as L2 L1 L0

  reg [2:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

    always @ (Q, M) begin
      case({Q,M})
        5'b00000:   next_Q = 3'b100;
        5'b00001:   next_Q = 3'b001;
        5'b00010:   next_Q = 3'b100;
        5'b00011:   next_Q = 3'b001;

        5'b00100:   next_Q = 3'b000;
        5'b00101:   next_Q = 3'b010;
        5'b00110:   next_Q = 3'b000;
        5'b00111:   next_Q = 3'b011;

        5'b01000:   next_Q = 3'b001;
        5'b01001:   next_Q = 3'b100;
        5'b01010:   next_Q = 3'b000;
        5'b01011:   next_Q = 3'b000;

        5'b01100:   next_Q = 3'b000;
        5'b01101:   next_Q = 3'b000;
        5'b01110:   next_Q = 3'b000;
        5'b01111:   next_Q = 3'b111;
```

Note: Here the output functions are merely the state variables – G=Q[2], Y=Q[1], R=Q[0]

```
        5'b10000:   next_Q = 3'b010;
        5'b10001:   next_Q = 3'b000;
        5'b10010:   next_Q = 3'b110;
        5'b10011:   next_Q = 3'b000;

        5'b10100:   next_Q = 3'b000;
        5'b10101:   next_Q = 3'b000;
        5'b10110:   next_Q = 3'b000;
        5'b10111:   next_Q = 3'b000;

        5'b11000:   next_Q = 3'b000;
        5'b11001:   next_Q = 3'b000;
        5'b11010:   next_Q = 3'b111;
        5'b11011:   next_Q = 3'b000;

        5'b11100:   next_Q = 3'b000;
        5'b11101:   next_Q = 3'b000;
        5'b11110:   next_Q = 3'b000;
        5'b11111:   next_Q = 3'b000;
      endcase
    end

endmodule
```

This realization uses 3 macrocells

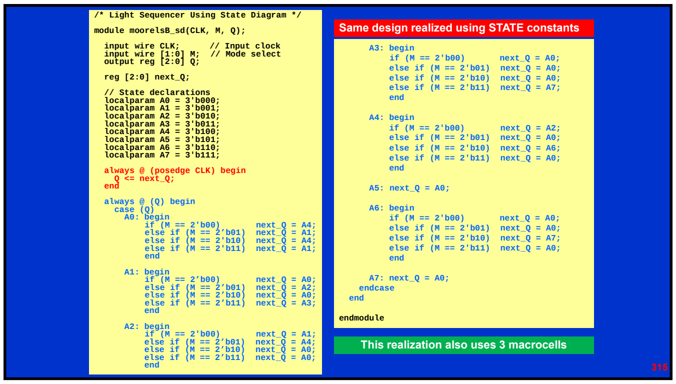
313



Assigning Names to States

0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left

314



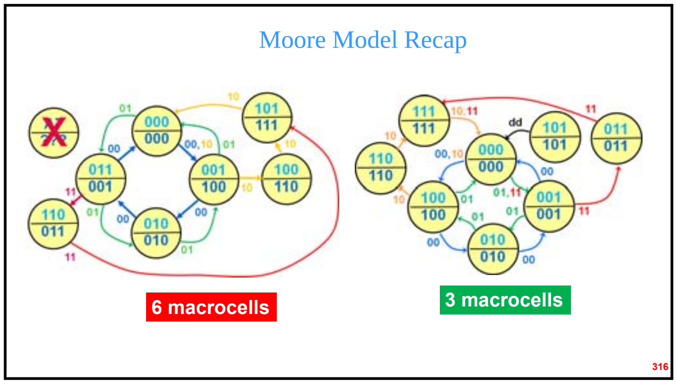
```
/* Light Sequencer Using State Diagram */

module moorelsB_sd(CLK, M, Q);

  input wire CLK;       // Input clock
  input wire [1:0] M;   // Mode select
  output reg [2:0] Q;

  reg [2:0] next_Q;

  // State declarations
  localparam A0 = 3'b000;
  localparam A1 = 3'b001;
  localparam A2 = 3'b010;
  localparam A3 = 3'b011;
  localparam A4 = 3'b100;
  localparam A5 = 3'b101;
  localparam A6 = 3'b110;
  localparam A7 = 3'b111;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q) begin
    case (Q)
      A0: begin
          if (M == 2'b00)       next_Q = A4;
          else if (M == 2’b01)  next_Q = A1;
          else if (M == 2'b10)  next_Q = A4;
          else if (M == 2'b11)  next_Q = A1;
          end

      A1: begin
          if (M == 2’b00)       next_Q = A0;
          else if (M == 2’b01)  next_Q = A2;
          else if (M == 2’b10)  next_Q = A0;
          else if (M == 2’b11)  next_Q = A3;
          end

      A2: begin
          if (M == 2'b00)       next_Q = A1;
          else if (M == 2’b01)  next_Q = A4;
          else if (M == 2’b10)  next_Q = A0;
          else if (M == 2’b11)  next_Q = A0;
          end
```

Same design realized using STATE constants

```
      A3: begin
          if (M == 2'b00)       next_Q = A0;
          else if (M == 2'b01)  next_Q = A0;
          else if (M == 2'b10)  next_Q = A0;
          else if (M == 2'b11)  next_Q = A7;
          end

      A4: begin
          if (M == 2'b00)       next_Q = A2;
          else if (M == 2'b01)  next_Q = A0;
          else if (M == 2'b10)  next_Q = A6;
          else if (M == 2'b11)  next_Q = A0;
          end

      A5: next_Q = A0;

      A6: begin
          if (M == 2'b00)       next_Q = A0;
          else if (M == 2'b01)  next_Q = A0;
          else if (M == 2'b10)  next_Q = A7;
          else if (M == 2'b11)  next_Q = A0;
          end

      A7: next_Q = A0;
    endcase
  end

endmodule
```

This realization also uses 3 macrocells

315

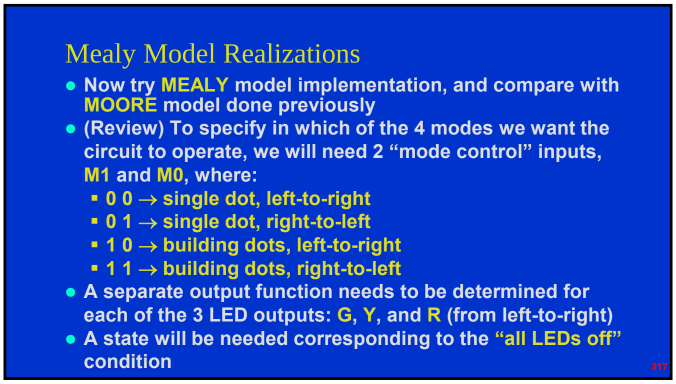


Moore Model Recap

6 macrocells

3 macrocells

316



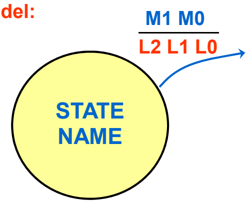
Mealy Model Realizations

- Now try MEALY model implementation, and compare with MOORE model done previously
- (Review) To specify in which of the 4 modes we want the circuit to operate, we will need 2 “mode control” inputs, M1 and M0, where:
  - 0 0 → single dot, left-to-right
  - 0 1 → single dot, right-to-left
  - 1 0 → building dots, left-to-right
  - 1 1 → building dots, right-to-left
- A separate output function needs to be determined for each of the 3 LED outputs: G, Y, and R (from left-to-right)
- A state will be needed corresponding to the “all LEDs off” condition

317

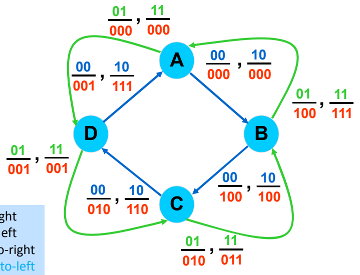
STEP 1: Construct a state transition diagram

Mealy Model:

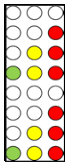


318

STEP 1: Construct a state transition diagram

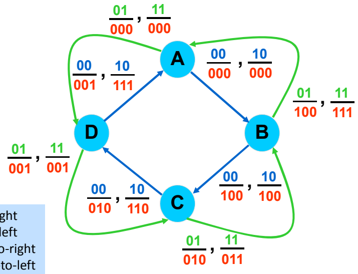


0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left



323

STEP 2: Minimize the number of states √
STEP 3: Assign state variable combinations

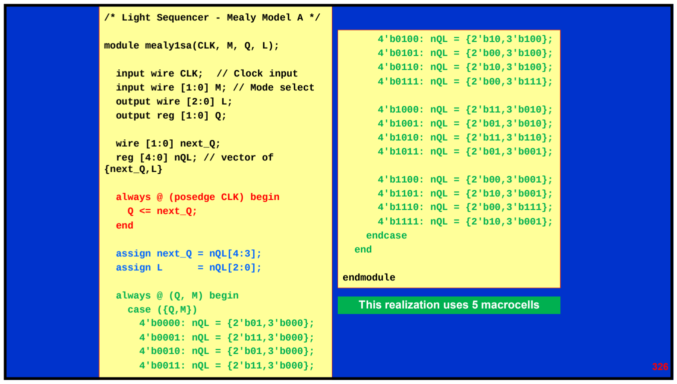


0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left

324

STEP 4: Construct a PS-NS/PO Table

PS		PI		NS		PO		
Q1	Q0	M1	M0	Q1*	Q0*	L2	L1	L0
0	0	0	0	0	1	0	0	0
		0	1	1	1	0	0	0
		1	0	0	1	0	0	0
		1	1	1	1	0	0	0
0	1	0	0	1	0	1	0	0
		0	1	0	0	1	0	0
		1	0	1	0	1	0	0
		1	1	0	0	1	1	1
1	0	0	0	1	1	0	1	0
		0	1	0	1	0	1	0
		1	0	1	1	1	1	0
		1	1	0	1	0	1	1
1	1	0	0	0	0	0	0	1
		0	1	1	0	0	0	1
		1	0	0	0	1	1	1
		1	1	1	0	0	0	1



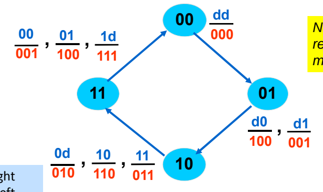
```
/* Light Sequencer - Mealy Model A */

module mealy1sa(CLK, M, Q, L);

  input wire CLK;   // Clock input
  input wire [1:0] M; // Mode select
  output wire [2:0] L;
  output reg [1:0] Q;

  wire [1:0] next_Q;
  reg [4:0] nQL; // vector of
{next_Q,L}

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  assign next_Q = nQL[4:3];
  assign L      = nQL[2:0];

  always @ (Q, M) begin
    case ({Q,M})
      4'b0000: nQL = {2'b01,3'b000};
      4'b0001: nQL = {2'b11,3'b000};
      4'b0010: nQL = {2'b01,3'b000};
      4'b0011: nQL = {2'b11,3'b000};
```

```
      4'b0100: nQL = {2'b10,3'b100};
      4'b0101: nQL = {2'b00,3'b100};
      4'b0110: nQL = {2'b10,3'b100};
      4'b0111: nQL = {2'b00,3'b111};

      4'b1000: nQL = {2'b11,3'b010};
      4'b1001: nQL = {2'b01,3'b010};
      4'b1010: nQL = {2'b11,3'b110};
      4'b1011: nQL = {2'b01,3'b001};

      4'b1100: nQL = {2'b00,3'b001};
      4'b1101: nQL = {2'b10,3'b001};
      4'b1110: nQL = {2'b00,3'b111};
      4'b1111: nQL = {2'b10,3'b001};
    endcase
  end

endmodule
```

This realization uses 5 macrocells

326

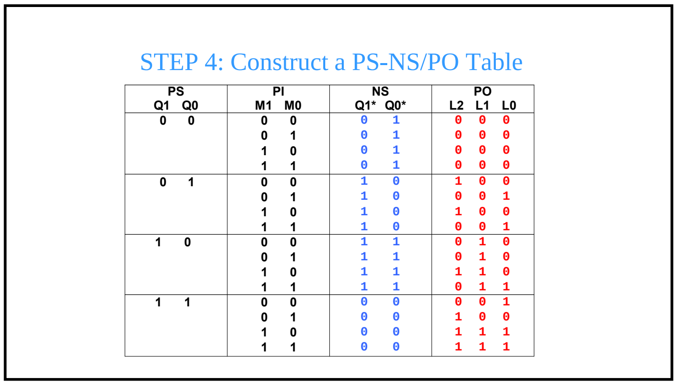
Revisit Steps 2 & 3: Did we pick the "best" state/output assignments possible?



Note there is no compelling reason to "reverse" the state machine "counting direction"

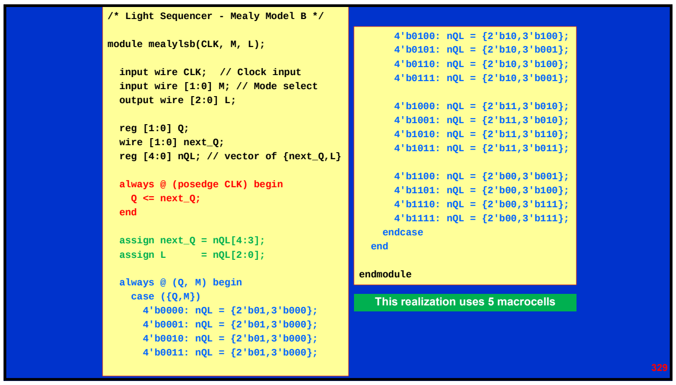
0 0 → single dot, left-to-right
0 1 → single dot, right-to-left
1 0 → building dots, left-to-right
1 1 → building dots, right-to-left

327



STEP 4: Construct a PS-NS/PO Table

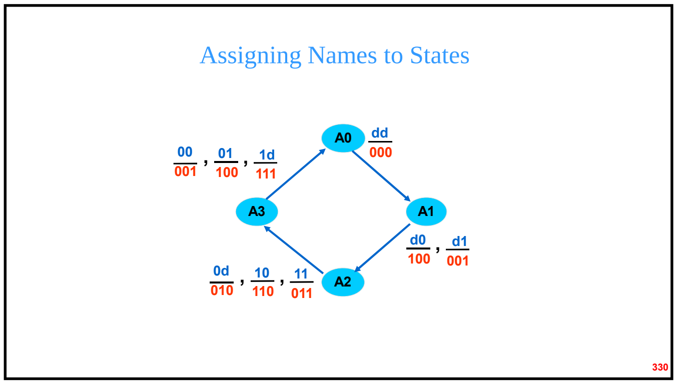
PS		PI		NS		PO		
Q1	Q0	M1	M0	Q1*	Q0*	L2	L1	L0
0	0	0	0	0	1	0	0	0
		0	1	0	1	0	0	0
		1	0	0	1	0	0	0
		1	1	0	1	0	0	0
0	1	0	0	1	0	1	0	0
		0	1	1	0	0	0	1
		1	0	1	0	1	0	0
		1	1	1	0	0	0	1
1	0	0	0	1	1	0	1	0
		0	1	1	1	0	1	0
		1	0	1	1	1	1	0
		1	1	1	1	0	1	1
1	1	0	0	0	0	0	0	1
		0	1	0	0	1	0	0
		1	0	0	0	1	1	1
		1	1	0	0	1	1	1



```
/* Light Sequencer - Mealy Model B */

module mealylsb(CLK, M, L);

  input wire CLK;  // Clock input
  input wire [1:0] M; // Mode select
  output wire [2:0] L;

  reg [1:0] Q;
  wire [1:0] next_Q;
  reg [4:0] nQL; // vector of {next_Q,L}

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  assign next_Q = nQL[4:3];
  assign L      = nQL[2:0];

  always @ (Q, M) begin
    case ({Q,M})
      4'b0000: nQL = {2'b01,3'b000};
      4'b0001: nQL = {2'b01,3'b000};
      4'b0010: nQL = {2'b01,3'b000};
      4'b0011: nQL = {2'b01,3'b000};
      4'b0100: nQL = {2'b10,3'b100};
      4'b0101: nQL = {2'b10,3'b001};
      4'b0110: nQL = {2'b10,3'b100};
      4'b0111: nQL = {2'b10,3'b001};

      4'b1000: nQL = {2'b11,3'b010};
      4'b1001: nQL = {2'b11,3'b010};
      4'b1010: nQL = {2'b11,3'b110};
      4'b1011: nQL = {2'b11,3'b011};

      4'b1100: nQL = {2'b00,3'b001};
      4'b1101: nQL = {2'b00,3'b100};
      4'b1110: nQL = {2'b00,3'b111};
      4'b1111: nQL = {2'b00,3'b111};
    endcase
  end

endmodule
```

This realization uses 5 macrocells

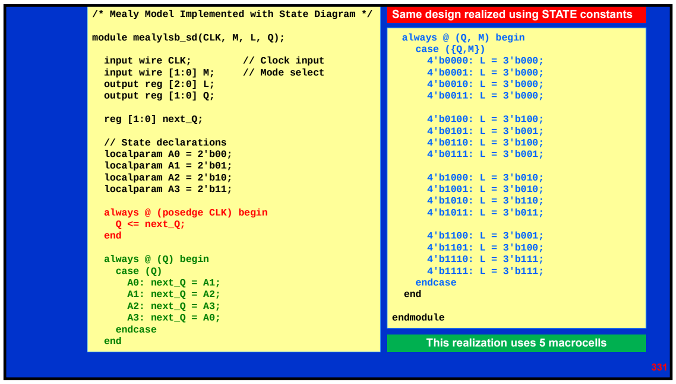
329



Assigning Names to States

A0
dd / 000
A1
d0 / 100 , d1 / 001
A2
0d / 010 , 10 / 110 , 11 / 011
A3
00 / 001 , 01 / 100 , 1d / 111

330



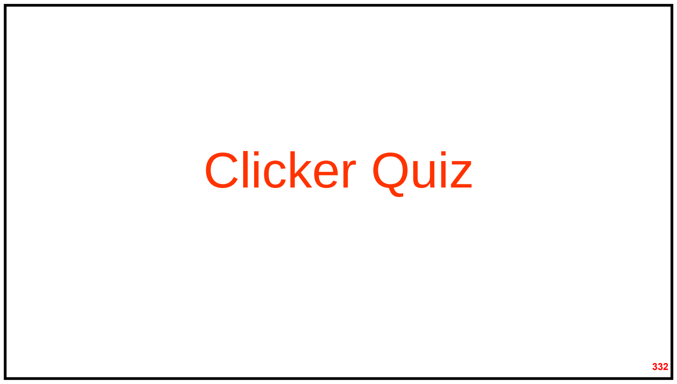
```
/* Mealy Model Implemented with State Diagram */

module mealylsb_sd(CLK, M, L, Q);

  input wire CLK;         // Clock input
  input wire [1:0] M;     // Mode select
  output reg [2:0] L;
  output reg [1:0] Q;

  reg [1:0] next_Q;

  // State declarations
  localparam A0 = 2'b00;
  localparam A1 = 2'b01;
  localparam A2 = 2'b10;
  localparam A3 = 2'b11;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q) begin
    case (Q)
      A0: next_Q = A1;
      A1: next_Q = A2;
      A2: next_Q = A3;
      A3: next_Q = A0;
    endcase
  end
```

Same design realized using STATE constants

```
  always @ (Q, M) begin
    case ({Q,M})
      4'b0000: L = 3'b000;
      4'b0001: L = 3'b000;
      4'b0010: L = 3'b000;
      4'b0011: L = 3'b000;

      4'b0100: L = 3'b100;
      4'b0101: L = 3'b001;
      4'b0110: L = 3'b100;
      4'b0111: L = 3'b001;

      4'b1000: L = 3'b010;
      4'b1001: L = 3'b010;
      4'b1010: L = 3'b110;
      4'b1011: L = 3'b011;

      4'b1100: L = 3'b001;
      4'b1101: L = 3'b100;
      4'b1110: L = 3'b111;
      4'b1111: L = 3'b111;
    endcase
  end

endmodule
```

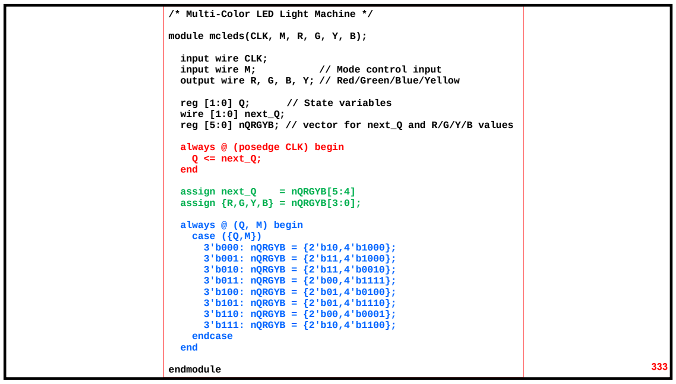
This realization uses 5 macrocells

331



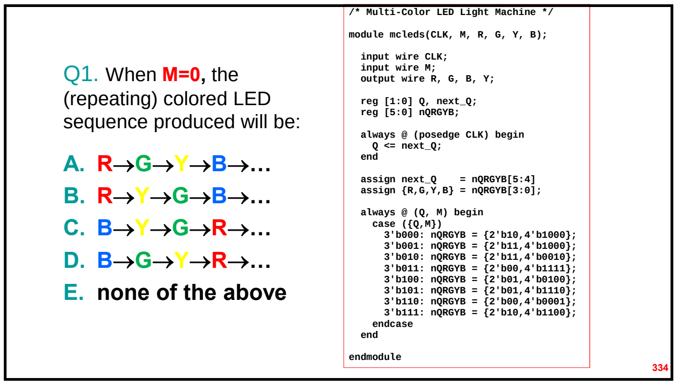
Clicker Quiz

332



```
/* Multi-Color LED Light Machine */

module mcleds(CLK, M, R, G, Y, B);

  input wire CLK;
  input wire M;           // Mode control input
  output wire R, G, B, Y; // Red/Green/Blue/Yellow

  reg [1:0] Q;      // State variables
  wire [1:0] next_Q;
  reg [5:0] nQRGYB; // vector for next_Q and R/G/Y/B values

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  assign next_Q    = nQRGYB[5:4]
  assign {R,G,Y,B} = nQRGYB[3:0];

  always @ (Q, M) begin
    case ({Q,M})
      3'b000: nQRGYB = {2'b10,4'b1000};
      3'b001: nQRGYB = {2'b11,4'b1000};
      3'b010: nQRGYB = {2'b11,4'b0010};
      3'b011: nQRGYB = {2'b00,4'b1111};
      3'b100: nQRGYB = {2'b01,4'b0100};
      3'b101: nQRGYB = {2'b01,4'b1110};
      3'b110: nQRGYB = {2'b00,4'b0001};
      3'b111: nQRGYB = {2'b10,4'b1100};
    endcase
  end

endmodule
```

333

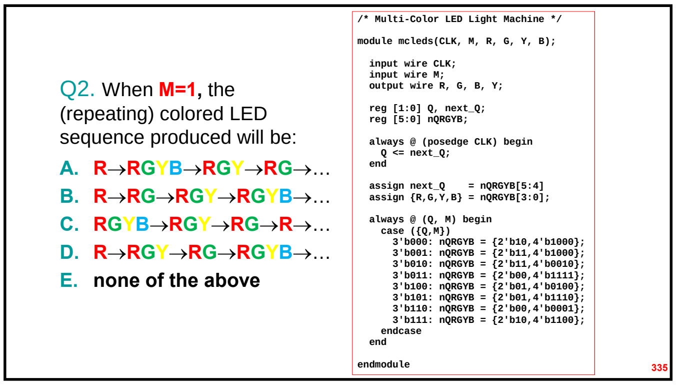


Q1. When **M=0**, the (repeating) colored LED sequence produced will be:

A. R→G→Y→B→…
B. R→Y→G→B→…
C. B→Y→G→R→…
D. B→G→Y→R→…
E. none of the above

```
/* Multi-Color LED Light Machine */

module mcleds(CLK, M, R, G, Y, B);

  input wire CLK;
  input wire M;
  output wire R, G, B, Y;

  reg [1:0] Q, next_Q;
  reg [5:0] nQRGYB;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  assign next_Q    = nQRGYB[5:4]
  assign {R,G,Y,B} = nQRGYB[3:0];

  always @ (Q, M) begin
    case ({Q,M})
      3'b000: nQRGYB = {2'b10,4'b1000};
      3'b001: nQRGYB = {2'b11,4'b1000};
      3'b010: nQRGYB = {2'b11,4'b0010};
      3'b011: nQRGYB = {2'b00,4'b1111};
      3'b100: nQRGYB = {2'b01,4'b0100};
      3'b101: nQRGYB = {2'b01,4'b1110};
      3'b110: nQRGYB = {2'b00,4'b0001};
      3'b111: nQRGYB = {2'b10,4'b1100};
    endcase
  end

endmodule
```

334

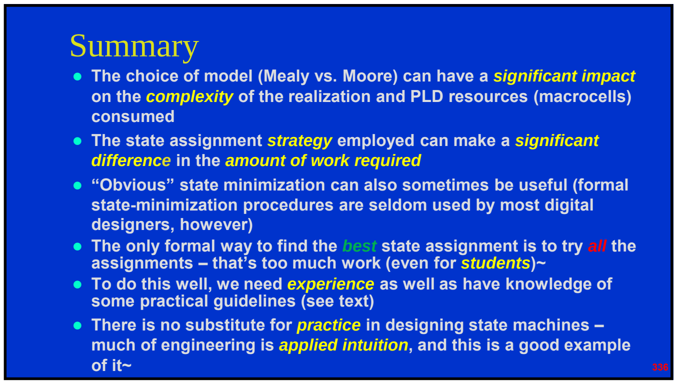


Q2. When **M=1**, the (repeating) colored LED sequence produced will be:

A. R→RGYB→RGY→RG→…
B. R→RG→RGY→RGYB→…
C. RGYB→RGY→RG→R→…
D. R→RGY→RG→RGYB→…
E. none of the above

```
/* Multi-Color LED Light Machine */

module mcleds(CLK, M, R, G, Y, B);

  input wire CLK;
  input wire M;
  output wire R, G, B, Y;

  reg [1:0] Q, next_Q;
  reg [5:0] nQRGYB;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  assign next_Q    = nQRGYB[5:4]
  assign {R,G,Y,B} = nQRGYB[3:0];

  always @ (Q, M) begin
    case ({Q,M})
      3'b000: nQRGYB = {2'b10,4'b1000};
      3'b001: nQRGYB = {2'b11,4'b1000};
      3'b010: nQRGYB = {2'b11,4'b0010};
      3'b011: nQRGYB = {2'b00,4'b1111};
      3'b100: nQRGYB = {2'b01,4'b0100};
      3'b101: nQRGYB = {2'b01,4'b1110};
      3'b110: nQRGYB = {2'b00,4'b0001};
      3'b111: nQRGYB = {2'b10,4'b1100};
    endcase
  end

endmodule
```

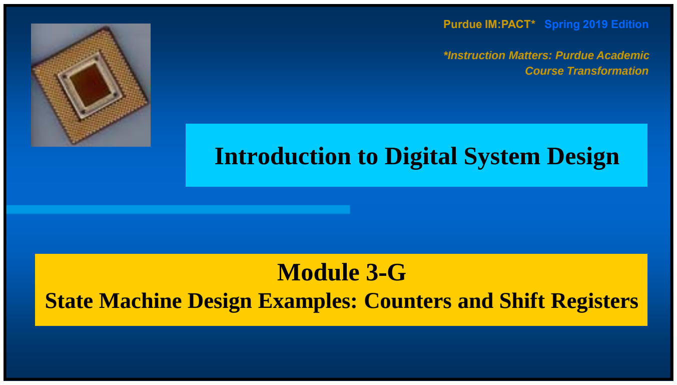
335



Summary

- The choice of model (Mealy vs. Moore) can have a *significant impact* on the *complexity* of the realization and PLD resources (macrocells) consumed
- The state assignment *strategy* employed can make a *significant difference* in the *amount of work required*
- "Obvious" state minimization can also sometimes be useful (formal state-minimization procedures are seldom used by most digital designers, however)
- The only formal way to find the *best* state assignment is to try *all* the assignments – that's too much work (even for *students*)~
- To do this well, we need *experience* as well as have knowledge of some practical guidelines (see text)
- There is no substitute for *practice* in designing state machines – much of engineering is *applied intuition*, and this is a good example of it~

336

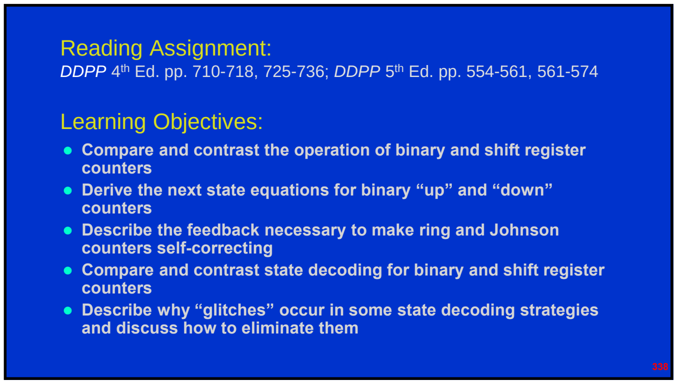


Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 3-G
State Machine Design Examples: Counters and Shift Registers

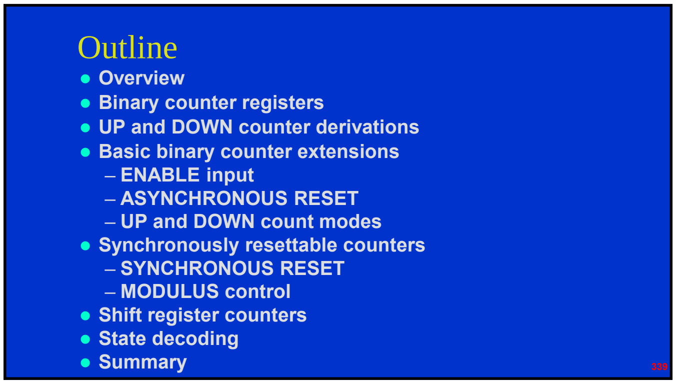


Reading Assignment:
DDPP 4th Ed. pp. 710-718, 725-736; *DDPP* 5th Ed. pp. 554-561, 561-574

Learning Objectives:

- Compare and contrast the operation of binary and shift register counters
- Derive the next state equations for binary "up" and "down" counters
- Describe the feedback necessary to make ring and Johnson counters self-correcting
- Compare and contrast state decoding for binary and shift register counters
- Describe why "glitches" occur in some state decoding strategies and discuss how to eliminate them

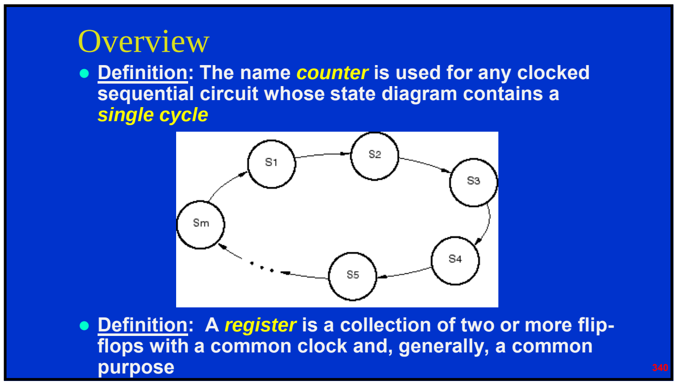
338



Outline

- Overview
- Binary counter registers
- UP and DOWN counter derivations
- Basic binary counter extensions
  - ENABLE input
  - ASYNCHRONOUS RESET
  - UP and DOWN count modes
- Synchronously resettable counters
  - SYNCHRONOUS RESET
  - MODULUS control
- Shift register counters
- State decoding
- Summary

339



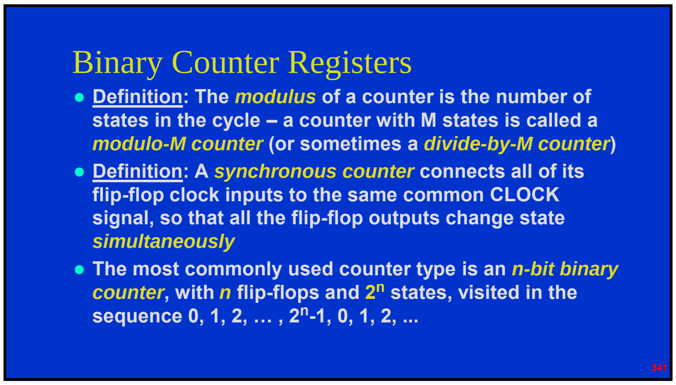
Overview

- Definition: The name *counter* is used for any clocked sequential circuit whose state diagram contains a *single cycle*

S1
S2
S3
S4
S5
Sm

- Definition: A *register* is a collection of two or more flip-flops with a common clock and, generally, a common purpose

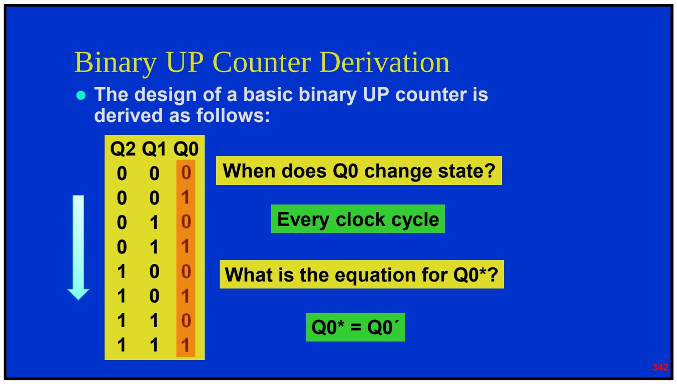
340



Binary Counter Registers

- Definition: The *modulus* of a counter is the number of states in the cycle – a counter with M states is called a *modulo-M counter* (or sometimes a *divide-by-M counter*)
- Definition: A *synchronous counter* connects all of its flip-flop clock inputs to the same common CLOCK signal, so that all the flip-flop outputs change state *simultaneously*
- The most commonly used counter type is an *n-bit binary counter*, with *n* flip-flops and 2^n states, visited in the sequence 0, 1, 2, … , 2^n-1, 0, 1, 2, ...

341



Binary UP Counter Derivation

- The design of a basic binary UP counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

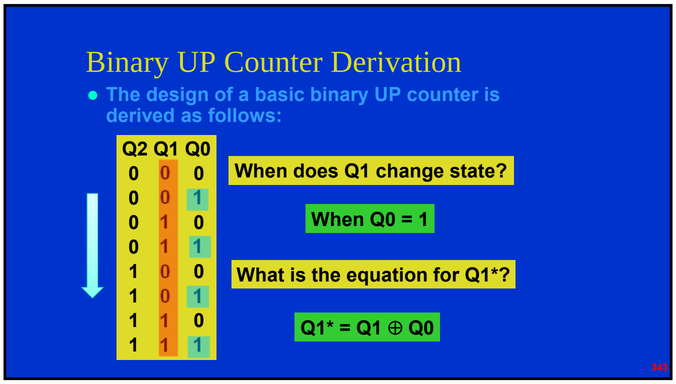
When does Q0 change state?

Every clock cycle

What is the equation for Q0*?

Q0* = Q0´

342



Binary UP Counter Derivation

- The design of a basic binary UP counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

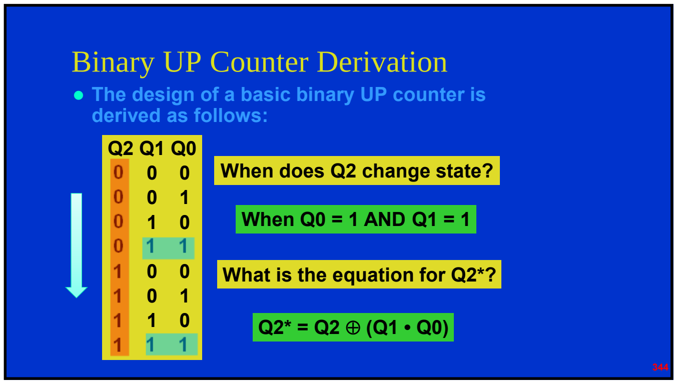
When does Q1 change state?

When Q0 = 1

What is the equation for Q1*?

Q1* = Q1 ⊕ Q0

343



Binary UP Counter Derivation

- The design of a basic binary UP counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

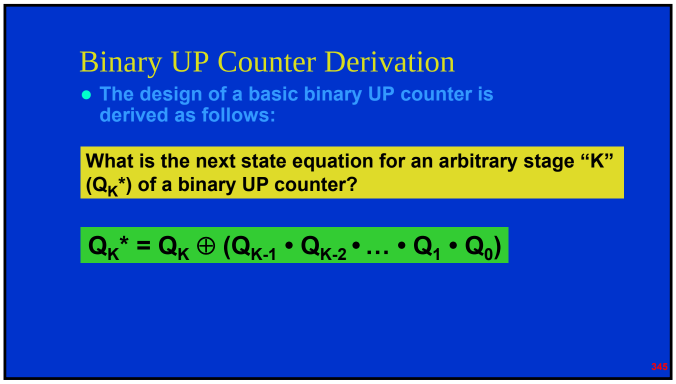
When does Q2 change state?

When Q0 = 1 AND Q1 = 1

What is the equation for Q2*?

Q2* = Q2 ⊕ (Q1 • Q0)

344



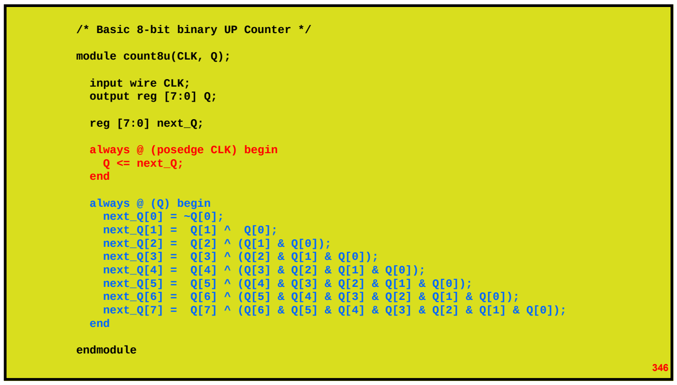
Binary UP Counter Derivation

- The design of a basic binary UP counter is derived as follows:

What is the next state equation for an arbitrary stage "K" (Q_K*) of a binary UP counter?

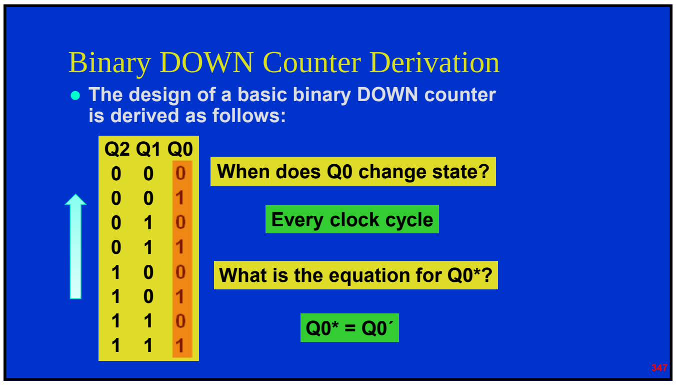
$$Q_K{}^* = Q_K \oplus (Q_{K-1} \bullet Q_{K-2} \bullet \ldots \bullet Q_1 \bullet Q_0)$$

345



```
/* Basic 8-bit binary UP Counter */

module count8u(CLK, Q);

  input wire CLK;
  output reg [7:0] Q;

  reg [7:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q) begin
    next_Q[0] = ~Q[0];
    next_Q[1] =  Q[1] ^  Q[0];
    next_Q[2] =  Q[2] ^ (Q[1] & Q[0]);
    next_Q[3] =  Q[3] ^ (Q[2] & Q[1] & Q[0]);
    next_Q[4] =  Q[4] ^ (Q[3] & Q[2] & Q[1] & Q[0]);
    next_Q[5] =  Q[5] ^ (Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    next_Q[6] =  Q[6] ^ (Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    next_Q[7] =  Q[7] ^ (Q[6] & Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
  end

endmodule
```

346



Binary DOWN Counter Derivation

- The design of a basic binary DOWN counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

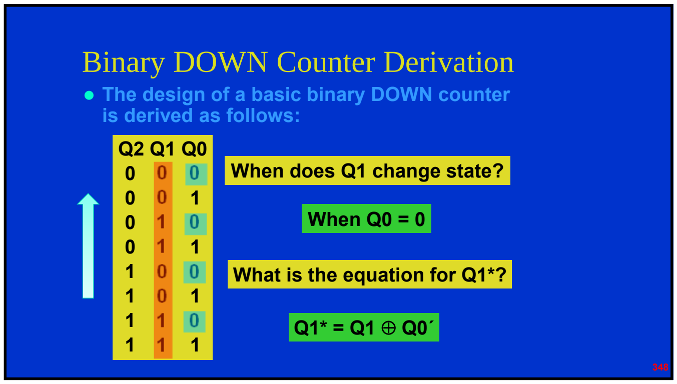
When does Q0 change state?

Every clock cycle

What is the equation for Q0*?

Q0* = Q0´

347



Binary DOWN Counter Derivation

- The design of a basic binary DOWN counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

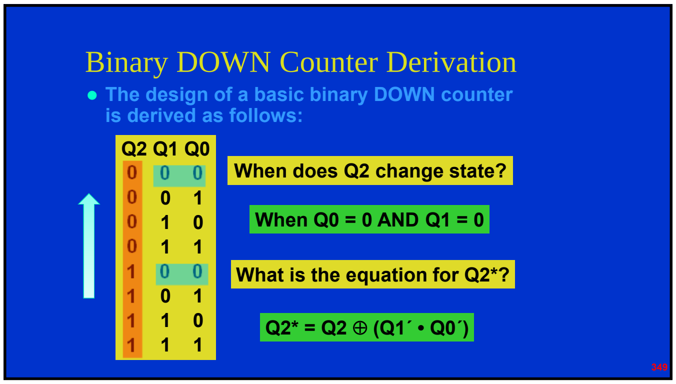
When does Q1 change state?

When Q0 = 0

What is the equation for Q1*?

$Q1^* = Q1 \oplus Q0'$

348



Binary DOWN Counter Derivation

- The design of a basic binary DOWN counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

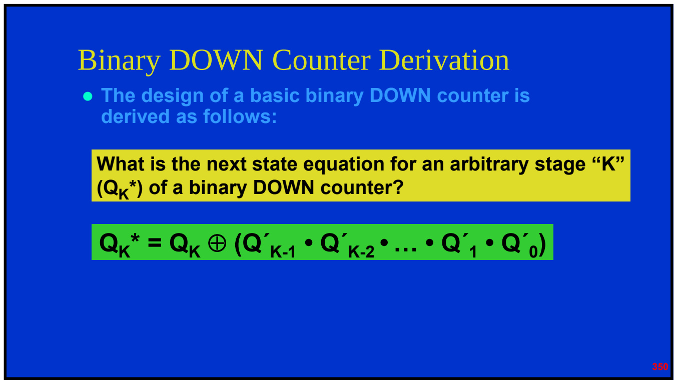
When does Q2 change state?

When Q0 = 0 AND Q1 = 0

What is the equation for Q2*?

$Q2^* = Q2 \oplus (Q1' \bullet Q0')$

349



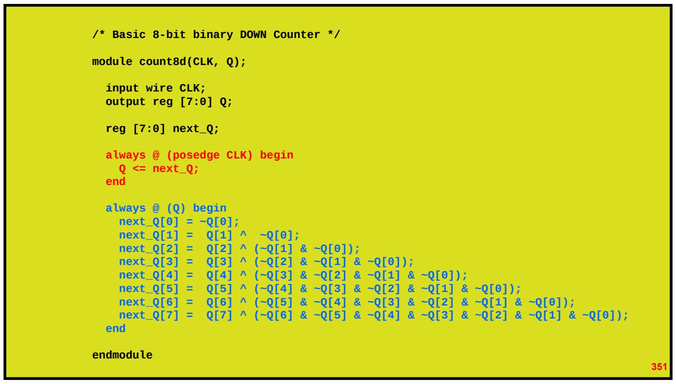
Binary DOWN Counter Derivation

- The design of a basic binary DOWN counter is derived as follows:

What is the next state equation for an arbitrary stage “K” (Q_K*) of a binary DOWN counter?

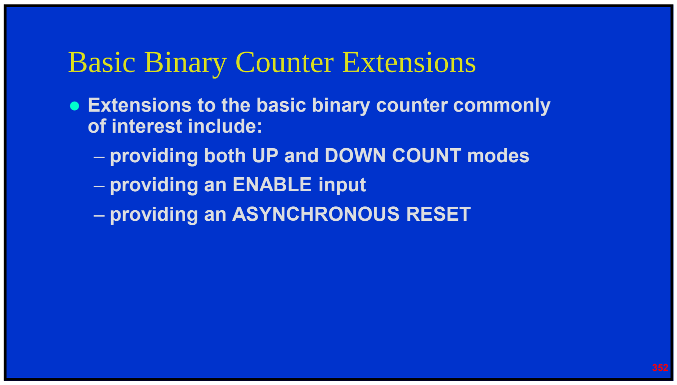
$$Q_K^* = Q_K \oplus (Q'_{K-1} \bullet Q'_{K-2} \bullet \ldots \bullet Q'_1 \bullet Q'_0)$$

350



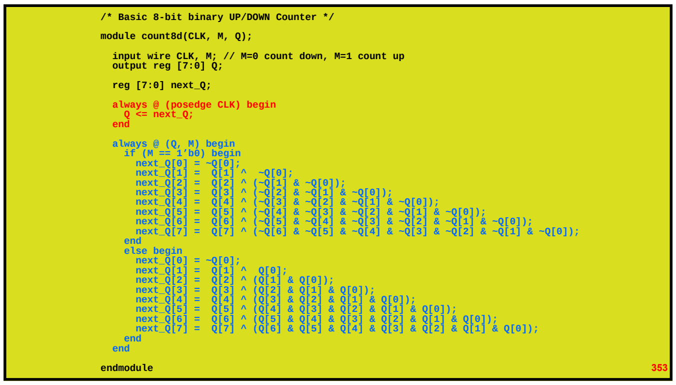
```
/* Basic 8-bit binary DOWN Counter */

module count8d(CLK, Q);

  input wire CLK;
  output reg [7:0] Q;

  reg [7:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q) begin
    next_Q[0] = ~Q[0];
    next_Q[1] =  Q[1] ^  ~Q[0];
    next_Q[2] =  Q[2] ^ (~Q[1] & ~Q[0]);
    next_Q[3] =  Q[3] ^ (~Q[2] & ~Q[1] & ~Q[0]);
    next_Q[4] =  Q[4] ^ (~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
    next_Q[5] =  Q[5] ^ (~Q[4] & ~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
    next_Q[6] =  Q[6] ^ (~Q[5] & ~Q[4] & ~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
    next_Q[7] =  Q[7] ^ (~Q[6] & ~Q[5] & ~Q[4] & ~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
  end

endmodule
```

351

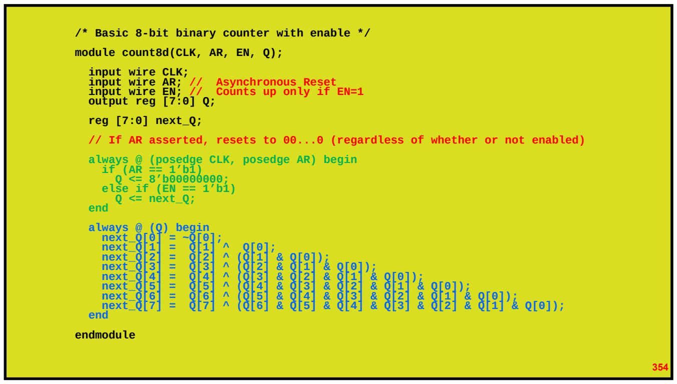


Basic Binary Counter Extensions

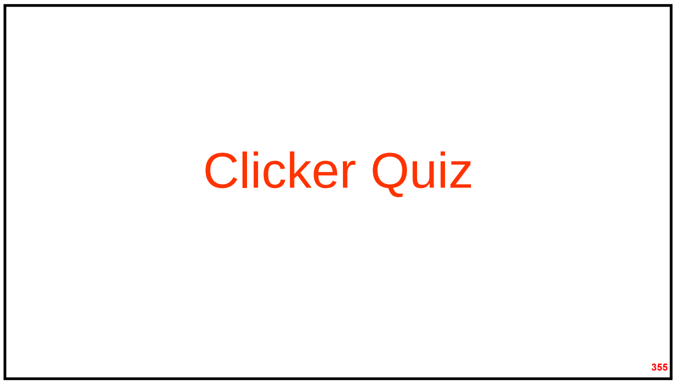
- Extensions to the basic binary counter commonly of interest include:
  - providing both UP and DOWN COUNT modes
  - providing an ENABLE input
  - providing an ASYNCHRONOUS RESET



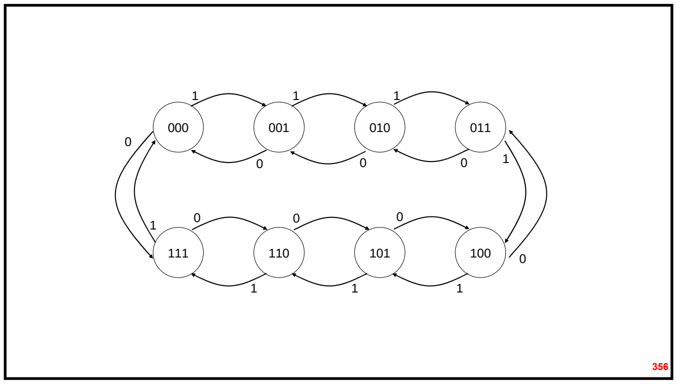
```
/* Basic 8-bit binary UP/DOWN Counter */

module count8d(CLK, M, Q);

  input wire CLK, M; // M=0 count down, M=1 count up
  output reg [7:0] Q;

  reg [7:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q, M) begin
    if (M == 1'b0) begin
      next_Q[0] = ~Q[0];
      next_Q[1] =  Q[1] ^  ~Q[0];
      next_Q[2] =  Q[2] ^ (~Q[1] & ~Q[0]);
      next_Q[3] =  Q[3] ^ (~Q[2] & ~Q[1] & ~Q[0]);
      next_Q[4] =  Q[4] ^ (~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
      next_Q[5] =  Q[5] ^ (~Q[4] & ~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
      next_Q[6] =  Q[6] ^ (~Q[5] & ~Q[4] & ~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
      next_Q[7] =  Q[7] ^ (~Q[6] & ~Q[5] & ~Q[4] & ~Q[3] & ~Q[2] & ~Q[1] & ~Q[0]);
    end
    else begin
      next_Q[0] = ~Q[0];
      next_Q[1] =  Q[1] ^  Q[0];
      next_Q[2] =  Q[2] ^ (Q[1] & Q[0]);
      next_Q[3] =  Q[3] ^ (Q[2] & Q[1] & Q[0]);
      next_Q[4] =  Q[4] ^ (Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[5] =  Q[5] ^ (Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[6] =  Q[6] ^ (Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[7] =  Q[7] ^ (Q[6] & Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    end
  end

endmodule
```

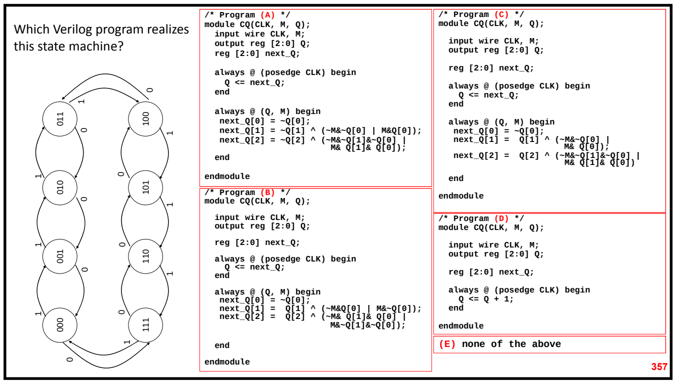


```
/* Basic 8-bit binary counter with enable */

module count8d(CLK, AR, EN, Q);

  input wire CLK;
  input wire AR; //  Asynchronous Reset
  input wire EN; //  Counts up only if EN=1
  output reg [7:0] Q;

  reg [7:0] next_Q;

  // If AR asserted, resets to 00...0 (regardless of whether or not enabled)

  always @ (posedge CLK, posedge AR) begin
    if (AR == 1'b1)
      Q <= 8'b00000000;
    else if (EN == 1'b1)
      Q <= next_Q;
  end

  always @ (Q) begin
    next_Q[0] = ~Q[0];
    next_Q[1] =  Q[1] ^  Q[0];
    next_Q[2] =  Q[2] ^ (Q[1] & Q[0]);
    next_Q[3] =  Q[3] ^ (Q[2] & Q[1] & Q[0]);
    next_Q[4] =  Q[4] ^ (Q[3] & Q[2] & Q[1] & Q[0]);
    next_Q[5] =  Q[5] ^ (Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    next_Q[6] =  Q[6] ^ (Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    next_Q[7] =  Q[7] ^ (Q[6] & Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
  end

endmodule
```



Clicker Quiz





Which Verilog program realizes this state machine?

```
/* Program (A) */
module CQ(CLK, M, Q);
  input wire CLK, M;
  output reg [2:0] Q;
  reg [2:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q, M) begin
   next_Q[0] = ~Q[0];
   next_Q[1] = ~Q[1] ^ (~M&~Q[0] | M&Q[0]);
   next_Q[2] = ~Q[2] ^ (~M&~Q[1]&~Q[0] |
                        M& Q[1]& Q[0]);
  end

endmodule
```

```
/* Program (B) */
module CQ(CLK, M, Q);

  input wire CLK, M;
  output reg [2:0] Q;

  reg [2:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q, M) begin
   next_Q[0] = ~Q[0];
   next_Q[1] =  Q[1] ^ (~M&Q[0] | M&~Q[0]);
   next_Q[2] =  Q[2] ^ (~M& Q[1]& Q[0] |
                        M&~Q[1]&~Q[0]);

  end

endmodule
```

```
/* Program (C) */
module CQ(CLK, M, Q);

  input wire CLK, M;
  output reg [2:0] Q;

  reg [2:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q, M) begin
   next_Q[0] = ~Q[0];
   next_Q[1] =  Q[1] ^ (~M&~Q[0] |
                        M& Q[0]);
   next_Q[2] =  Q[2] ^ (~M&~Q[1]&~Q[0] |
                        M& Q[1]& Q[0])

  end

endmodule
```

```
/* Program (D) */
module CQ(CLK, M, Q);

  input wire CLK, M;
  output reg [2:0] Q;

  reg [2:0] next_Q;

  always @ (posedge CLK) begin
    Q <= Q + 1;
  end

endmodule
```

(E) none of the above

Resettable Counters

- In addition to an *asynchronous reset* (which allows to the counter to be placed in a known initial state), it is sometimes useful to provide a *synchronous reset* capability
- Such a counter is useful in applications where the number of states in the counting sequence is determined *dynamically*
- Example: State counter in a computer's execute unit, where the number of cycles necessary to complete an instruction varies
- Another variation: Counter with a "programmable" final state (modulo M)

359

```
/* Resettable 8-bit binary UP Counter */

module rcnt8U(CLK, R, Q);

  input wire CLK;
  input wire R;      // Synchronous Reset
  output reg [7:0] Q;
  reg [7:0] next_Q;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  // If R  = 1, counter resets to 0 on the next clock edge
  always @ (Q) begin
    if (R == 1'b1) begin
      next_Q = 8'b00000000;
    end
    else begin
      next_Q[0] = ~Q[0];
      next_Q[1] =  Q[1] ^  Q[0];
      next_Q[2] =  Q[2] ^ (Q[1] & Q[0]);
      next_Q[3] =  Q[3] ^ (Q[2] & Q[1] & Q[0]);
      next_Q[4] =  Q[4] ^ (Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[5] =  Q[5] ^ (Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[6] =  Q[6] ^ (Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[7] =  Q[7] ^ (Q[6] & Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    end
  end

endmodule
```

360

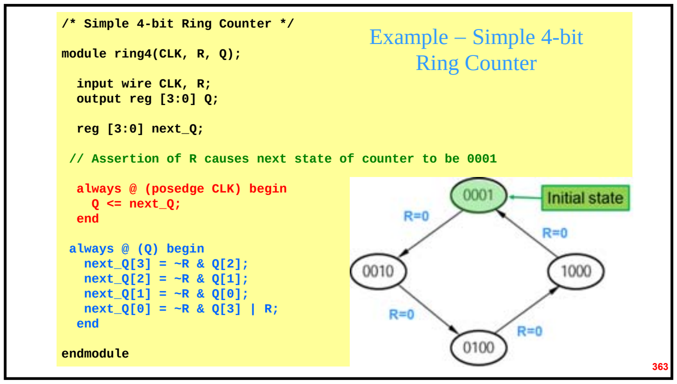
```
/* Modulo Resettable 8-bit binary UP Counter */

module mrcnt8U(CLK, MOD, Q);

  input wire CLK;
  input wire [7:0] MOD;  // MOD value
  output reg [7:0] Q;

  reg [7:0] next_Q;
  wire R;

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  // Count up to value on MOD → 00..00 → value up to MOD
  assign R = (Q == MOD);

  always @ (Q) begin
    if (R == 1'b1) begin
      next_Q = 8'b00000000;  // When Q reaches MOD, reset to 0
    end
    else begin
      next_Q[0] = ~Q[0];
      next_Q[1] =  Q[1] ^  Q[0];
      next_Q[2] =  Q[2] ^ (Q[1] & Q[0]);
      next_Q[3] =  Q[3] ^ (Q[2] & Q[1] & Q[0]);
      next_Q[4] =  Q[4] ^ (Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[5] =  Q[5] ^ (Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[6] =  Q[6] ^ (Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
      next_Q[7] =  Q[7] ^ (Q[6] & Q[5] & Q[4] & Q[3] & Q[2] & Q[1] & Q[0]);
    end
  end

endmodule
```

361

Shift-Register Counters

- Definition: A shift register whose state diagram is *cyclic* is called a *shift-register counter*
- Unlike a binary counter, a shift-register counter does not count in an "up" or "down" binary sequence, but is useful in many "control" applications nonetheless
- The simplest shift-register counter uses an n-bit shift register to obtain a counter with n states, and is called a *ring counter*
- A ring counter sequence is sometimes referred to as *"one hot"*

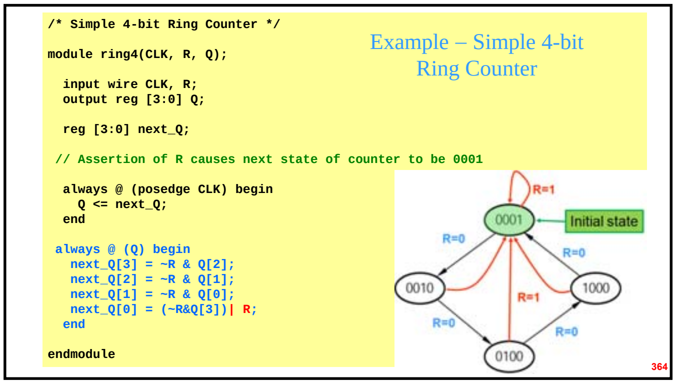
362



Example – Simple 4-bit Ring Counter

```
/* Simple 4-bit Ring Counter */

module ring4(CLK, R, Q);

  input wire CLK, R;
  output reg [3:0] Q;

  reg [3:0] next_Q;

 // Assertion of R causes next state of counter to be 0001

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

 always @ (Q) begin
   next_Q[3] = ~R & Q[2];
   next_Q[2] = ~R & Q[1];
   next_Q[1] = ~R & Q[0];
   next_Q[0] = ~R & Q[3] | R;
  end

endmodule
```

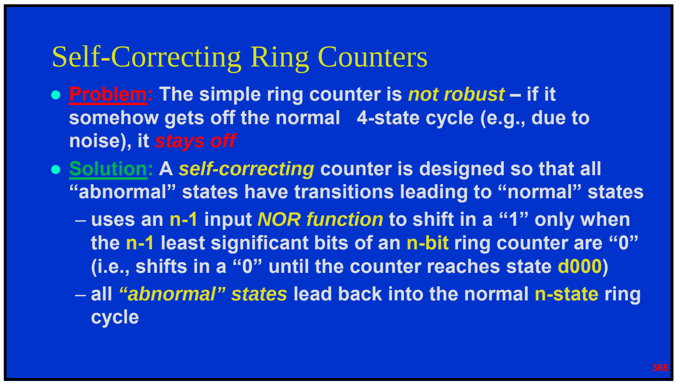
363



Example – Simple 4-bit Ring Counter

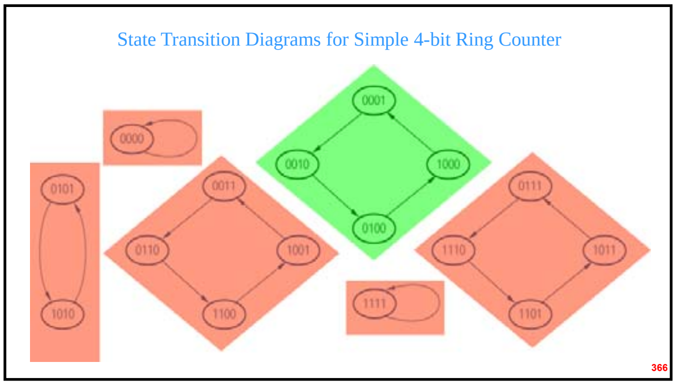
```
/* Simple 4-bit Ring Counter */

module ring4(CLK, R, Q);

  input wire CLK, R;
  output reg [3:0] Q;

  reg [3:0] next_Q;

 // Assertion of R causes next state of counter to be 0001

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

 always @ (Q) begin
   next_Q[3] = ~R & Q[2];
   next_Q[2] = ~R & Q[1];
   next_Q[1] = ~R & Q[0];
   next_Q[0] = (~R&Q[3])| R;
  end

endmodule
```

364

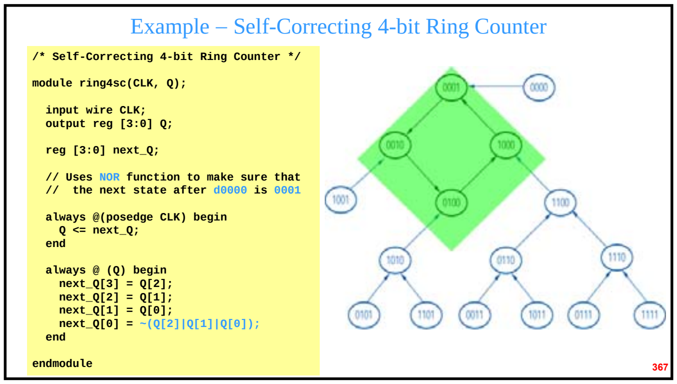


Self-Correcting Ring Counters

- Problem: The simple ring counter is *not robust* – if it somehow gets off the normal 4-state cycle (e.g., due to noise), it *stays off*
- Solution: A *self-correcting* counter is designed so that all "abnormal" states have transitions leading to "normal" states
  - uses an n-1 input *NOR function* to shift in a "1" only when the n-1 least significant bits of an n-bit ring counter are "0" (i.e., shifts in a "0" until the counter reaches state d000)
  - all *"abnormal" states* lead back into the normal n-state ring cycle

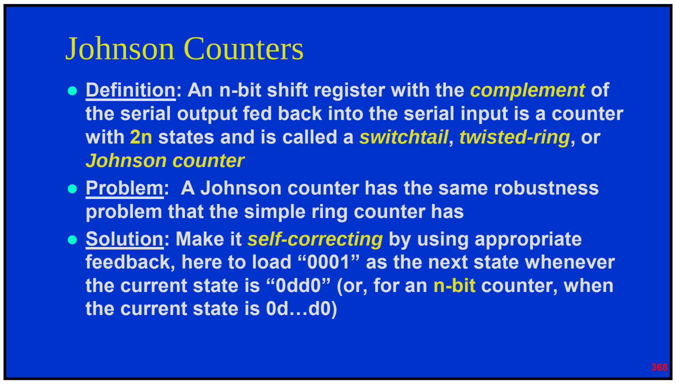


State Transition Diagrams for Simple 4-bit Ring Counter



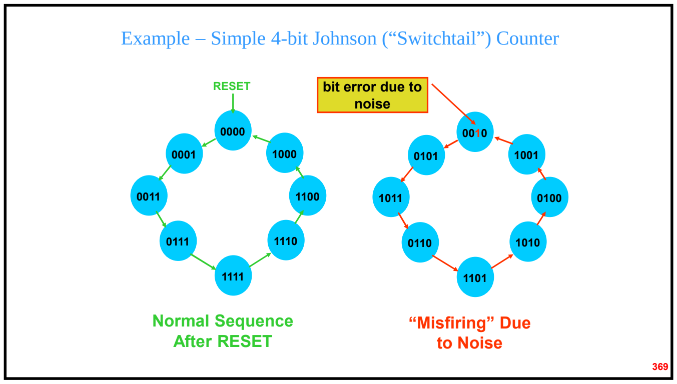
Example – Self-Correcting 4-bit Ring Counter

```
/* Self-Correcting 4-bit Ring Counter */

module ring4sc(CLK, Q);

  input wire CLK;
  output reg [3:0] Q;

  reg [3:0] next_Q;

  // Uses NOR function to make sure that
  //  the next state after d0000 is 0001

  always @(posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q) begin
    next_Q[3] = Q[2];
    next_Q[2] = Q[1];
    next_Q[1] = Q[0];
    next_Q[0] = ~(Q[2]|Q[1]|Q[0]);
  end

endmodule
```



Johnson Counters

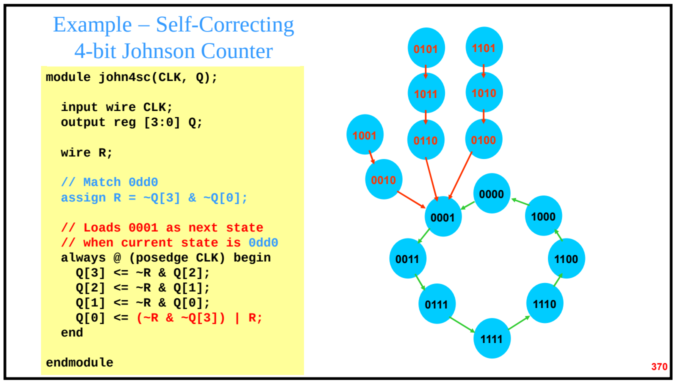
- Definition: An n-bit shift register with the *complement* of the serial output fed back into the serial input is a counter with 2n states and is called a *switchtail*, *twisted-ring*, or *Johnson counter*
- Problem: A Johnson counter has the same robustness problem that the simple ring counter has
- Solution: Make it *self-correcting* by using appropriate feedback, here to load "0001" as the next state whenever the current state is "0dd0" (or, for an n-bit counter, when the current state is 0d…d0)



Example – Simple 4-bit Johnson ("Switchtail") Counter

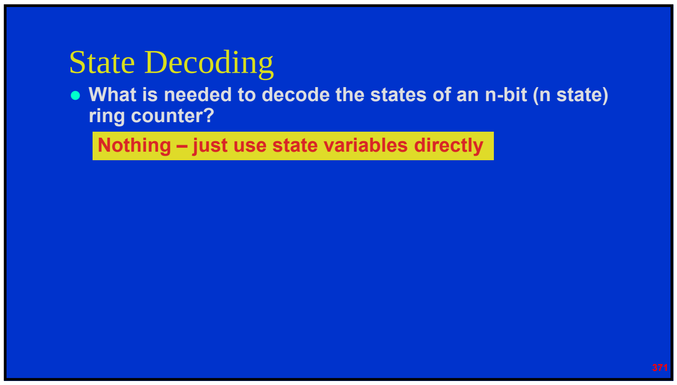
Normal Sequence After RESET

"Misfiring" Due to Noise



Example – Self-Correcting 4-bit Johnson Counter

```
module john4sc(CLK, Q);

  input wire CLK;
  output reg [3:0] Q;

  wire R;

  // Match 0dd0
  assign R = ~Q[3] & ~Q[0];

  // Loads 0001 as next state
  // when current state is 0dd0
  always @ (posedge CLK) begin
    Q[3] <= ~R & Q[2];
    Q[2] <= ~R & Q[1];
    Q[1] <= ~R & Q[0];
    Q[0] <= (~R & ~Q[3]) | R;
  end

endmodule
```

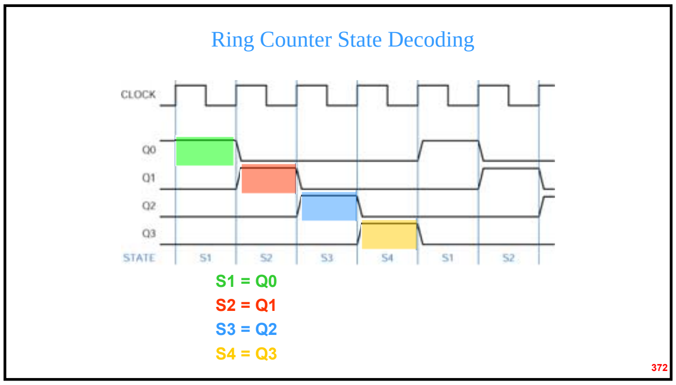


State Decoding

- What is needed to decode the states of an n-bit (n state) ring counter?

  Nothing – just use state variables directly

371



Ring Counter State Decoding

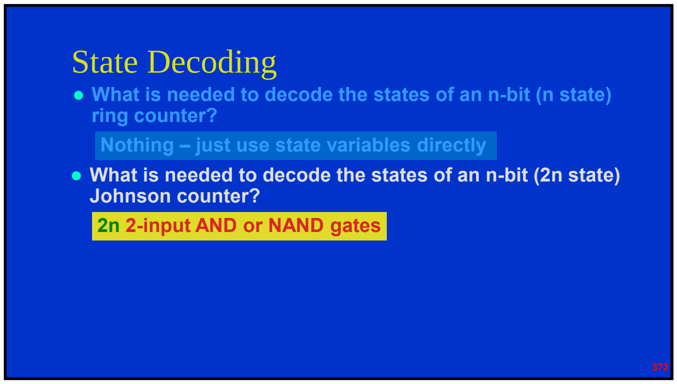
S1 = Q0

S2 = Q1

S3 = Q2

S4 = Q3

372



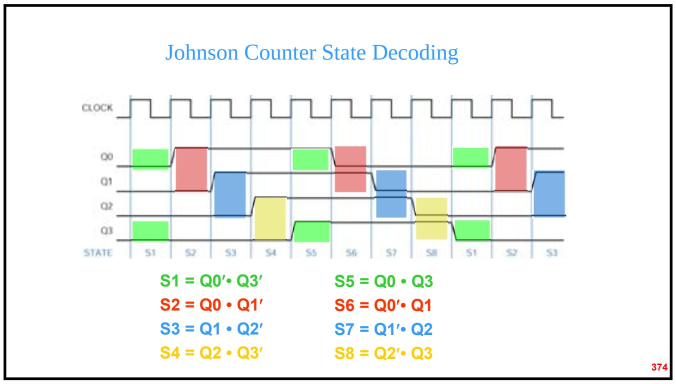
State Decoding

- What is needed to decode the states of an n-bit (n state) ring counter?

  Nothing – just use state variables directly
- What is needed to decode the states of an n-bit (2n state) Johnson counter?

  2n 2-input AND or NAND gates

373



Johnson Counter State Decoding

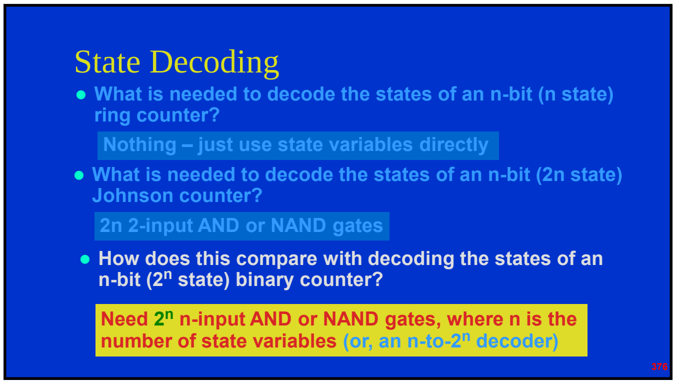
S1 = Q0′• Q3′	S5 = Q0 • Q3
S2 = Q0 • Q1′	S6 = Q0′• Q1
S3 = Q1 • Q2′	S7 = Q1′• Q2
S4 = Q2 • Q3′	S8 = Q2′• Q3

374

```
/* Self-correcting 4-bit Johnson counter with decoded output */

module john4scd(CLK, S);

  input wire CLK;
  output reg [7:0] S;       // Decoded output

  wire R;
  reg [3:0] Q, next_Q;

  assign R = ~Q[3] & ~Q[0]; // Match 0xx0

  always @ (posedge CLK) begin
    Q <= next_Q;
  end

  always @ (Q) begin
    next_Q[3] = ~R & Q[2];
    next_Q[2] = ~R & Q[1];
    next_Q[1] = ~R & Q[0];
    next_Q[0] = (~R & ~Q[3]) | R;

    S[0] = ~Q[3] & ~Q[0];
    S[1] = ~Q[1] &  Q[0];
    S[2] = ~Q[2] &  Q[1];
    S[3] = ~Q[3] &  Q[2];
    S[4] =  Q[3] &  Q[0];
    S[5] =  Q[1] & ~Q[0];
    S[6] =  Q[2] & ~Q[1];
    S[7] =  Q[3] & ~Q[2];
  end

endmodule
```

state decoding

375



State Decoding

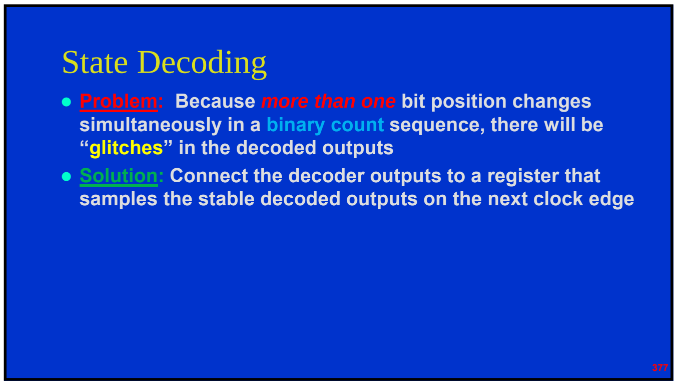
- What is needed to decode the states of an n-bit (n state) ring counter?

  Nothing – just use state variables directly
- What is needed to decode the states of an n-bit (2n state) Johnson counter?

  2n 2-input AND or NAND gates
- How does this compare with decoding the states of an n-bit (2^n state) binary counter?

  Need 2^n n-input AND or NAND gates, where n is the number of state variables (or, an n-to-2^n decoder)

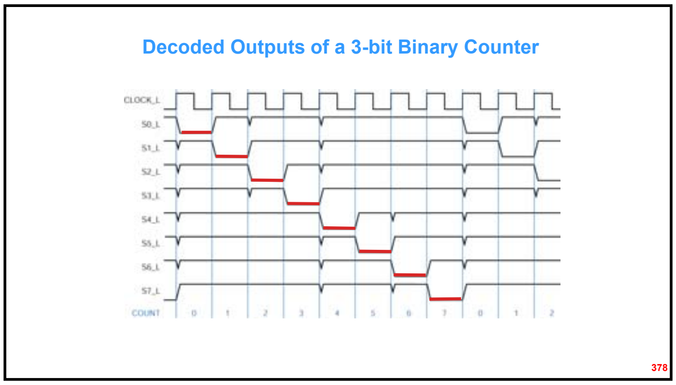
376



State Decoding

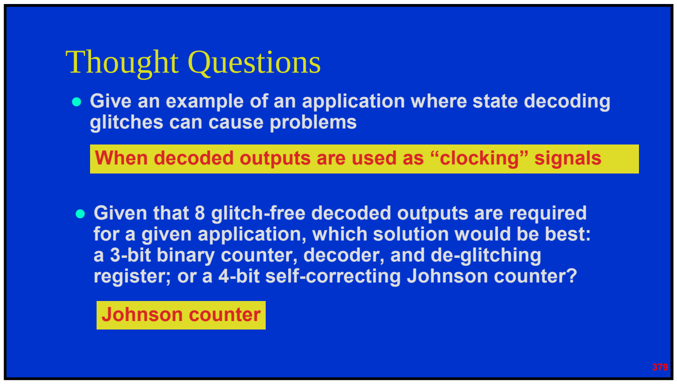
- Problem: Because *more than one* bit position changes simultaneously in a binary count sequence, there will be "glitches" in the decoded outputs
- Solution: Connect the decoder outputs to a register that samples the stable decoded outputs on the next clock edge

377



Decoded Outputs of a 3-bit Binary Counter

378



Thought Questions

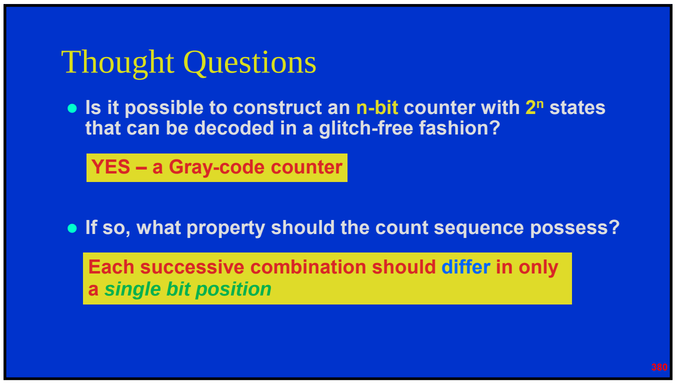
- Give an example of an application where state decoding glitches can cause problems

When decoded outputs are used as "clocking" signals

- Given that 8 glitch-free decoded outputs are required for a given application, which solution would be best: a 3-bit binary counter, decoder, and de-glitching register; or a 4-bit self-correcting Johnson counter?

Johnson counter

379



Thought Questions

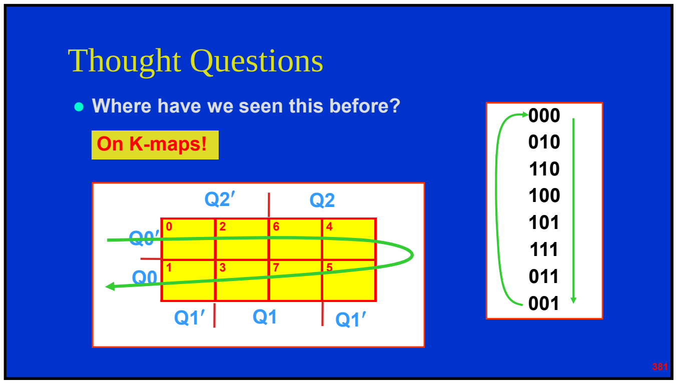
- Is it possible to construct an n-bit counter with 2^n states that can be decoded in a glitch-free fashion?

YES – a Gray-code counter

- If so, what property should the count sequence possess?

Each successive combination should differ in only a *single bit position*

380



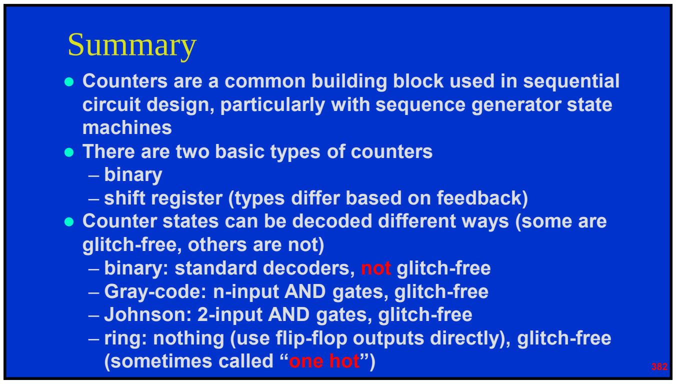
Thought Questions

- Where have we seen this before?

On K-maps!

000
010
110
100
101
111
011
001

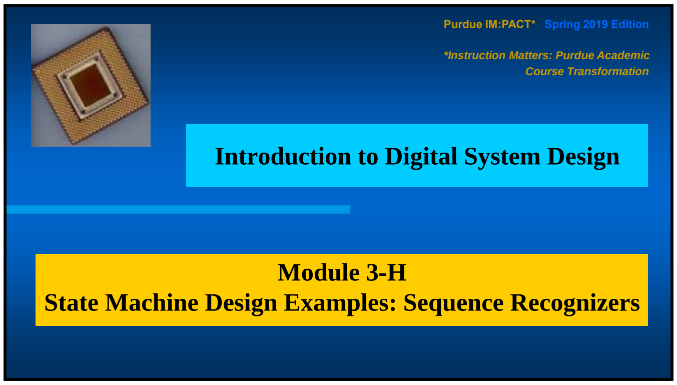
381



Summary

- Counters are a common building block used in sequential circuit design, particularly with sequence generator state machines
- There are two basic types of counters
  - binary
  - shift register (types differ based on feedback)
- Counter states can be decoded different ways (some are glitch-free, others are not)
  - binary: standard decoders, not glitch-free
  - Gray-code: n-input AND gates, glitch-free
  - Johnson: 2-input AND gates, glitch-free
  - ring: nothing (use flip-flop outputs directly), glitch-free (sometimes called "one hot")

382

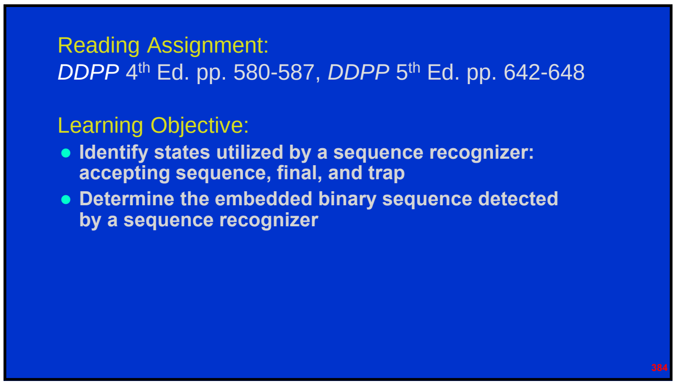


Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 3-H
State Machine Design Examples: Sequence Recognizers

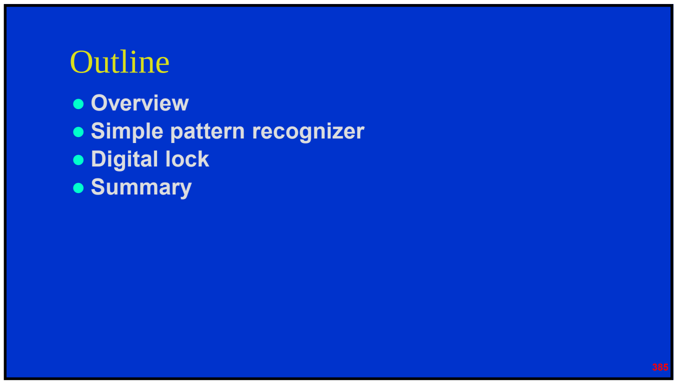


Reading Assignment:
DDPP 4th Ed. pp. 580-587, *DDPP* 5th Ed. pp. 642-648

Learning Objective:

- **Identify states utilized by a sequence recognizer: accepting sequence, final, and trap**
- **Determine the embedded binary sequence detected by a sequence recognizer**

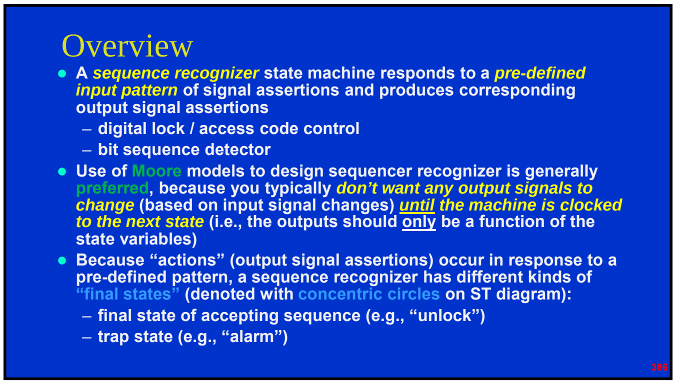
384



Outline

- **Overview**
- **Simple pattern recognizer**
- **Digital lock**
- **Summary**

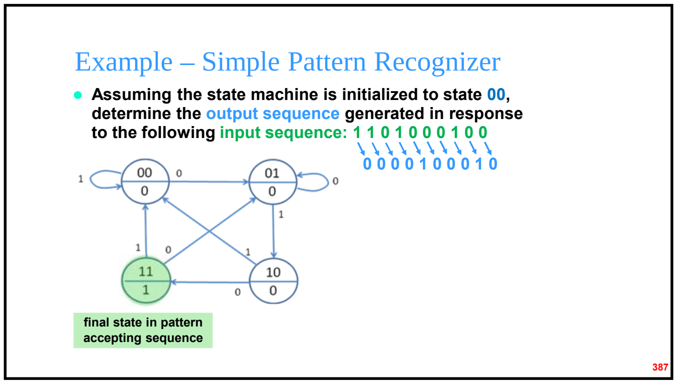
385



Overview

- **A *sequence recognizer* state machine responds to a *pre-defined input pattern* of signal assertions and produces corresponding output signal assertions**
  - **digital lock / access code control**
  - **bit sequence detector**
- **Use of Moore models to design sequencer recognizer is generally preferred, because you typically *don't want any output signals to change* (based on input signal changes) *__until__ the machine is clocked to the next state* (i.e., the outputs should __only__ be a function of the state variables)**
- **Because "actions" (output signal assertions) occur in response to a pre-defined pattern, a sequence recognizer has different kinds of "final states" (denoted with concentric circles on ST diagram):**
  - **final state of accepting sequence (e.g., "unlock")**
  - **trap state (e.g., "alarm")**

386



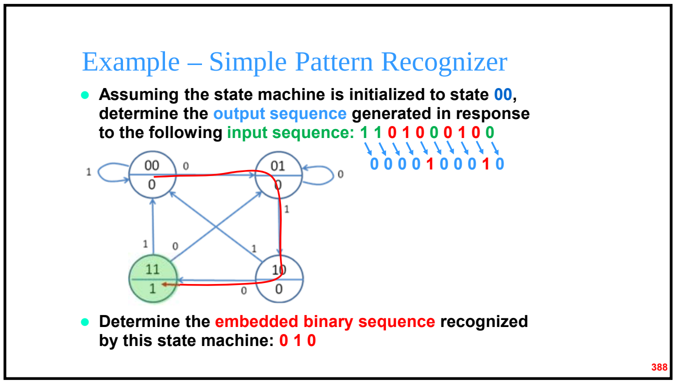
Example – Simple Pattern Recognizer

- **Assuming the state machine is initialized to state 00, determine the output sequence generated in response to the following input sequence: 1 1 0 1 0 0 0 1 0 0**

0 0 0 0 1 0 0 0 1 0

final state in pattern accepting sequence

387



Example – Simple Pattern Recognizer

- **Assuming the state machine is initialized to state 00, determine the output sequence generated in response to the following input sequence: 1 1 0 1 0 0 0 1 0 0**

0 0 0 0 1 0 0 0 1 0

- **Determine the embedded binary sequence recognized by this state machine: 0 1 0**

388

Example - Digital Combination Lock

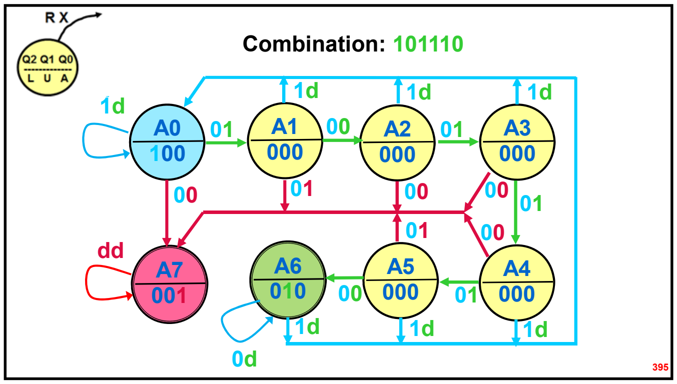
- Design a digital combination lock
  - unlocks when a fixed combination (binary sequence) is entered: 101110
  - has three inputs:
    - X – combination data
    - R – relock / reset
    - RESET – asynchronous reset
  - has three output signals:
    - LOCKED
    - UNLOCKED
    - ALARM

389

Example - Digital Combination Lock

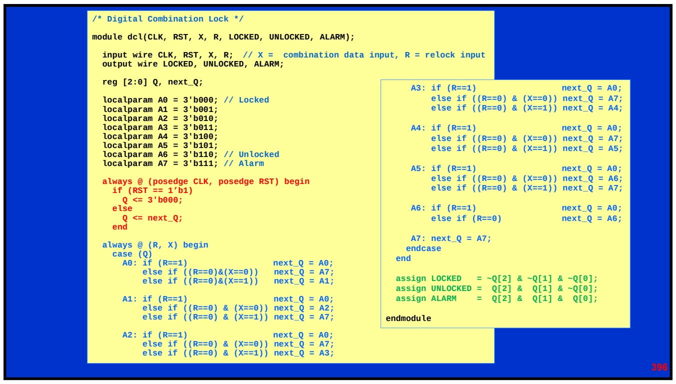
- Implement using Moore model
  - will need an initial "locked" state
  - will need six states to accept digits of combination (the last is "unlocked")
  - will need an "alarm" state
  - total number of states is eight; therefore, can implement with three state variables
- Types of states
  - accepting sequence (entering combination)
  - final state (sequence correctly entered)
  - trap state (error made while entering combination)

390



Combination: 101110

395



```
/* Digital Combination Lock */

module dcl(CLK, RST, X, R, LOCKED, UNLOCKED, ALARM);

  input wire CLK, RST, X, R;  // X =  combination data input, R = relock input
  output wire LOCKED, UNLOCKED, ALARM;

  reg [2:0] Q, next_Q;

  localparam A0 = 3'b000; // Locked
  localparam A1 = 3'b001;
  localparam A2 = 3'b010;
  localparam A3 = 3'b011;
  localparam A4 = 3'b100;
  localparam A5 = 3'b101;
  localparam A6 = 3'b110; // Unlocked
  localparam A7 = 3'b111; // Alarm

  always @ (posedge CLK, posedge RST) begin
    if (RST == 1'b1)
      Q <= 3'b000;
    else
      Q <= next_Q;
    end

  always @ (R, X) begin
    case (Q)
      A0: if (R==1)                  next_Q = A0;
          else if ((R==0)&(X==0))   next_Q = A7;
          else if ((R==0)&(X==1))   next_Q = A1;

      A1: if (R==1)                  next_Q = A0;
          else if ((R==0) & (X==0)) next_Q = A2;
          else if ((R==0) & (X==1)) next_Q = A7;

      A2: if (R==1)                  next_Q = A0;
          else if ((R==0) & (X==0)) next_Q = A7;
          else if ((R==0) & (X==1)) next_Q = A3;

      A3: if (R==1)                  next_Q = A0;
          else if ((R==0) & (X==0)) next_Q = A7;
          else if ((R==0) & (X==1)) next_Q = A4;

      A4: if (R==1)                  next_Q = A0;
          else if ((R==0) & (X==0)) next_Q = A7;
          else if ((R==0) & (X==1)) next_Q = A5;

      A5: if (R==1)                  next_Q = A0;
          else if ((R==0) & (X==0)) next_Q = A6;
          else if ((R==0) & (X==1)) next_Q = A7;

      A6: if (R==1)                  next_Q = A0;
          else if (R==0)             next_Q = A6;

      A7: next_Q = A7;
    endcase
  end

  assign LOCKED   = ~Q[2] & ~Q[1] & ~Q[0];
  assign UNLOCKED =  Q[2] &  Q[1] & ~Q[0];
  assign ALARM    =  Q[2] &  Q[1] &  Q[0];

endmodule
```

396

Summary

- A sequence recognizer is a state machine that produces output signal assertions in response to an input pattern
- Output signal assertions typically occur when the machine enters a "final state" associated with the accepting sequence
- Sequence recognizers are typically realized with Moore models, to prevent "spurious" behavior that might occur if the machine's outputs could potentially change in response to an input signal change without clocking it

397