Lecture Summary – Module 3

Sequential Logic Circuits

Learning Outcome: ***an ability to analyze and design sequential logic circuits***

Learning Objectives:

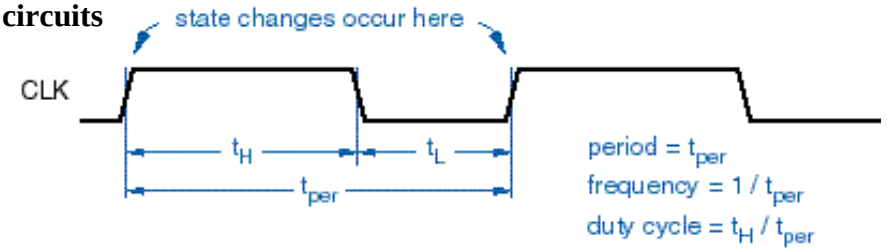
3-1. describe the difference between a combinational logic circuit and a sequential logic circuit
3-2. describe the difference between a feedback sequential circuit and a clocked synchronous state machine
3-3. define the state of a sequential circuit
3-4. define active high and active low as it pertains to clocking signals
3-5. define clock frequency and duty cycle
3-6. describe the operation of a bi-stable and analyze its behavior
3-7. define metastability and illustrate how the existence of a metastable equilibrium point can lead to a random next state
3-8. write present state – next state (PS-NS) equations that describes the behavior of a sequential circuit
3-9. draw a state transition diagram that depicts the behavior of a sequential circuit
3-10. construct a timing chart that depicts the behavior of a sequential circuit
3-11. draw a circuit for a set-reset ("S-R") latch and analyze its behavior
3-12. discuss what is meant by "transparent" (or "data following") in reference to the response of a latch
3-13. draw a circuit for an edge-triggered data ("D") flip-flop and analyze its behavior
3-14. compare the response of a latch and a flip-flop to the same set of stimuli
3-15. define setup and hold time and determine their nominal values from a timing chart
3-16. determine the frequency and duty cycle of a clocking signal
3-17. identify latch and flip-flop propagation delay paths and determine their values from a timing chart
3-18. describe the operation of a toggle ("T") flip-flop and analyze its behavior
3-19. derive a characteristic equation for any type of latch or flip-flop
3-20. identify the key elements of a clocked synchronous state machine: next state logic, state memory (flip-flops), and output logic
3-21. differentiate between Mealy and Moore model state machines, and draw a block diagram of each
3-22. analyze a clocked synchronous state machine realized as either a Mealy or Moore model
3-23. outline the steps required for state machine synthesis
3-24. derive an excitation table for any type of flip-flop
3-25. discuss reasons why formal state-minimization procedures are seldom used by experienced digital designers
3-26. draw block diagrams for Moore and Mealy type state machines and explain how each block can be coded in Verilog
3-27. draw a circuit for an oscillator and calculate its frequency of operation
3-28. draw a circuit for a bounce-free switch based on an S-R latch and analyze its behavior
3-29. design a clocked synchronous state machine and verify its operation
3-30. define minimum risk and minimum cost state machine design strategies, and discuss the tradeoffs between the two approaches
3-31. compare state assignment strategy and state machine model choice (Mealy vs. Moore) with respect to PLD resources (P-terms and macrocells) required for realization
3-32. compare and contrast the operation of binary and shift register counters
3-33. derive the next state equations for binary "up" and "down" counters
3-34. describe the feedback necessary to make ring and Johnson counters self-correcting
3-35. compare and contrast state decoding for binary and shift register counters
3-36. describe why "glitches" occur in some state decoding strategies and discuss how to eliminate them
3-37. identify states utilized by a sequence recognizer: accepting sequence, final, and trap
3-38. determine the embedded binary sequence detected by a sequence recognizer

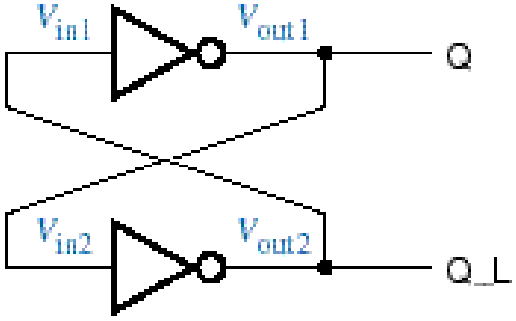
Lecture Summary – Module 3-A

Bistable Elements

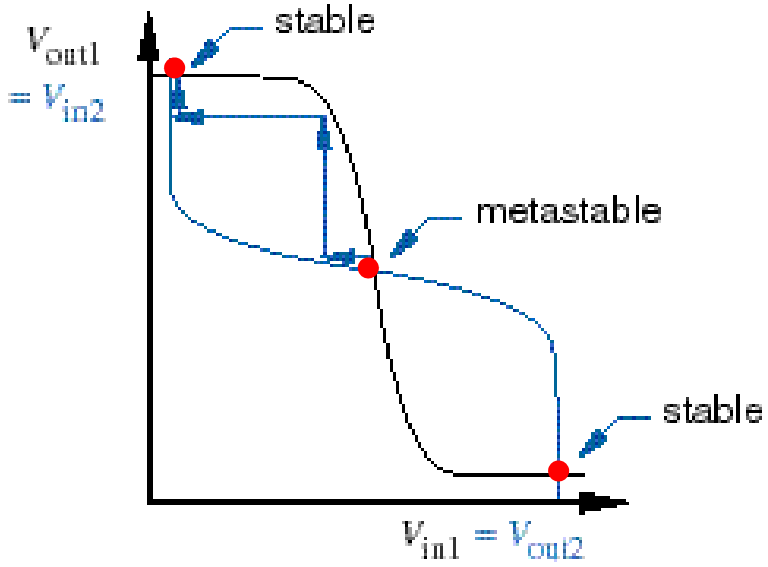
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 521-526

- **overview**
  - **combinational vs. sequential circuits**
  - **state of sequential circuit**
  - **finite state machine**
  - **clock signal**
    - **assertion level**
    - **period / frequency**
    - **duty cycle**
  - **types of sequential circuits**
    - **feedback**
    - **clocked synchronous**





- **bistable elements**
  - **"simplest" sequential circuit**
  - **no inputs (no way of controlling/changing state)**
  - **randomly powers up into one state or the other**
  - **digital analysis: two stable states**
  - **single state variable (Q)**
  - **analog analysis: additional quasi-stable state (metastable)**



Transfer functions ("inverter"):

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

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

Equilibrium points:

$V_{in1} = V_{out2}$

$V_{in2} = V_{out1}$

Random noise drives circuit to stable operating point

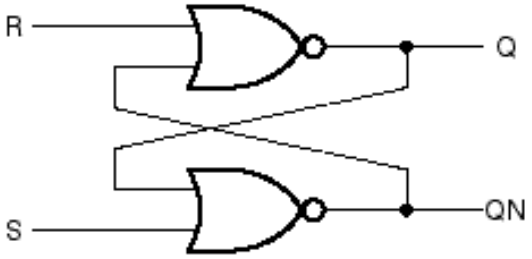
- **metastable behavior**
  - **comparable to dropping ball onto smooth hill**
  - **speed with which ball rolls to one side or the other depends on location it "hits"**
  - **important: if "simplest" sequential circuit is susceptible to metastable behavior, then clearly ALL sequential circuits are(!)**

Lecture Summary – Module 3-B

The Set-Reset (S-R) Latch

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 526-532

- **latches and flip-flops**
  - **flip-flop changes state based on *clocking signal***
  - **latch changes its output any time it is *enabled***
- **set-reset (S-R) latch**
  - **change bistable into latch by "adding an input" to each inverter (NOR gate)**
  - **two inputs**
    - **asserting S "sets" the latch state (Q output) to 1**
    - **asserting R "resets" the latch state to 0**
    - **if both S and R are negated, circuit behaves like bistable (retains its state)**
    - **if both S and R are asserted and then negated simultaneously, random next state**
- **exercise: construct a timing chart for the NOR-implemented S-R latch**
  - **assume each gate has delay τ**
  - **write the next state equations for Q and QN**

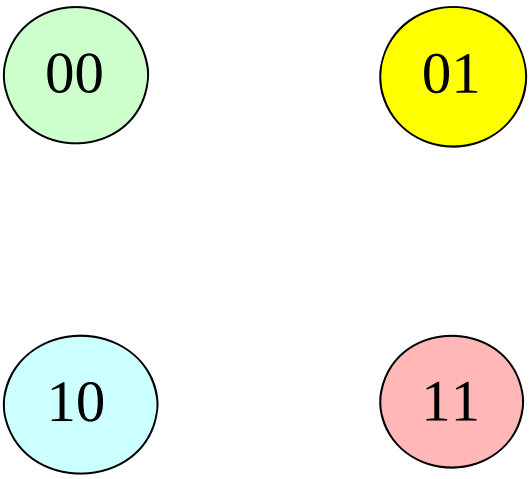


Q(t+τ) =

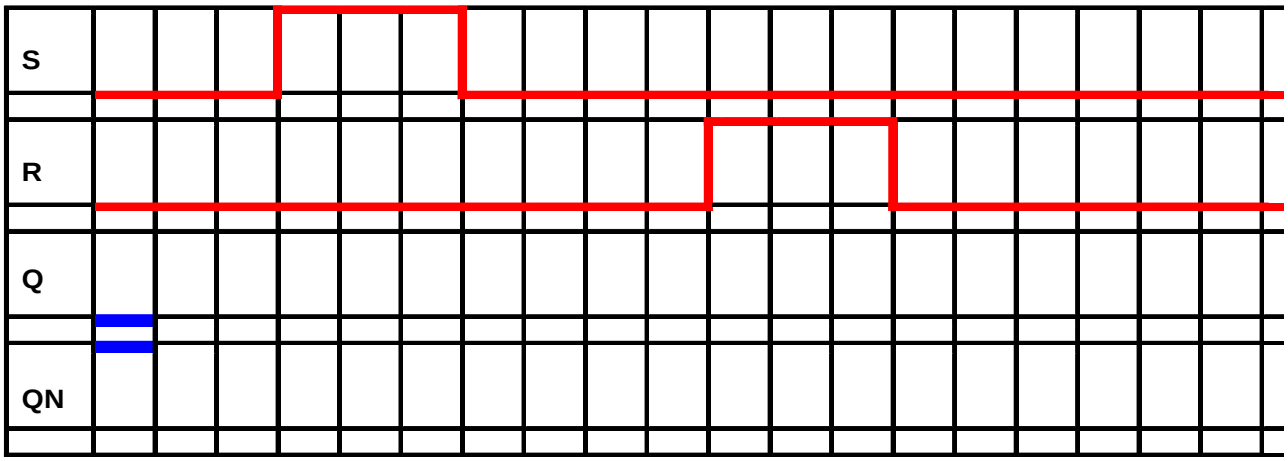
QN(t+τ) =

  - **create a present state – next state (PS-NS) table and state transition diagram (STD)**

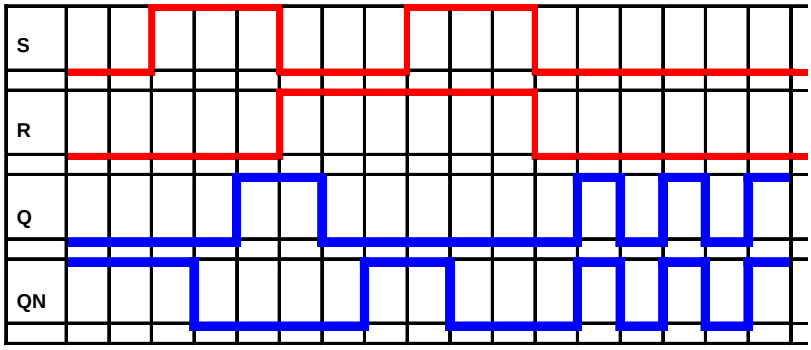
Present State		Present Input		Next State	
Q(t)	QN(t)	S(t)	R(t)	Q(t+τ)	QN(T+τ)
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1		
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

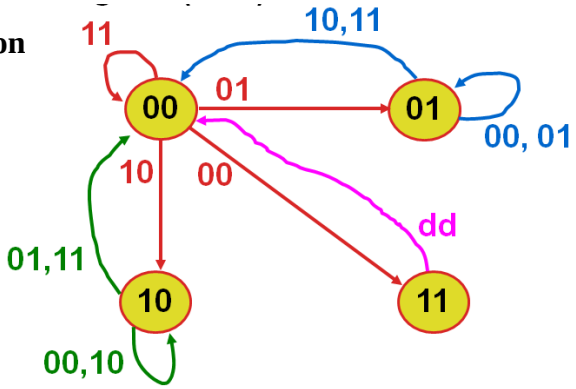


- **exercise, continued…**
  - **construct a timing chart based on the initial conditions and given inputs**



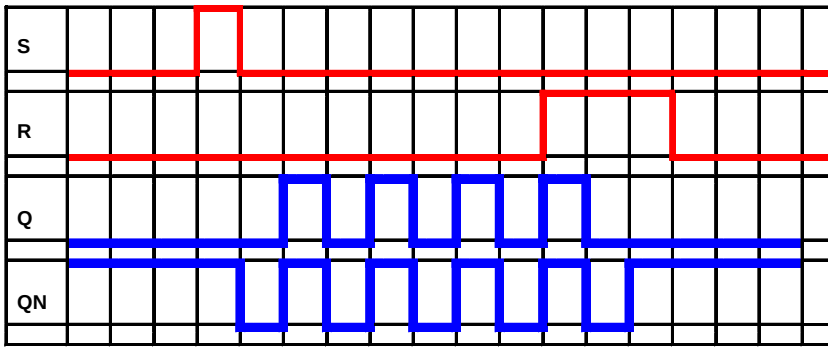
- **exercise: investigate response to the "1-1" input combination**





Present State		Present Input		Next State	
Q(t)	QN(t)	S(t)	R(t)	Q(t+τ)	QN(T+τ)
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

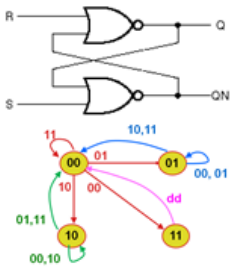
- **exercise: investigate response to a "glitch"**



- **propagation delay – time for an output to respond to an input transition**
  - **need to specify "path"**
  - **example: $t_{pLH(S\rightarrow Q)}$ is the rise propagation delay of the Q output in response to assertion of the S input**
  - **note that rise and fall propagation delays are typically *different***
- **minimum pulse width requirement (see "glitch" timing chart)**

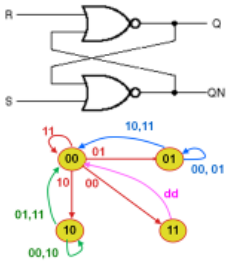
Q1. For the NOR-implemented SR latch, the following output combination **cannot occur at any time:**

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



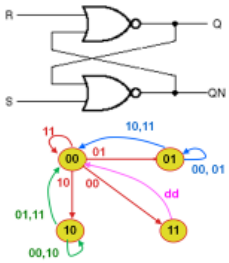
Q2. If the **input** combination **S=0, R=1** is applied to this circuit, the (steady state) output will be:

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



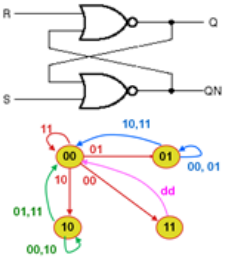
Q3. If the **input** combination **S=1, R=0** is applied to this circuit, the (steady state) output will be:

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

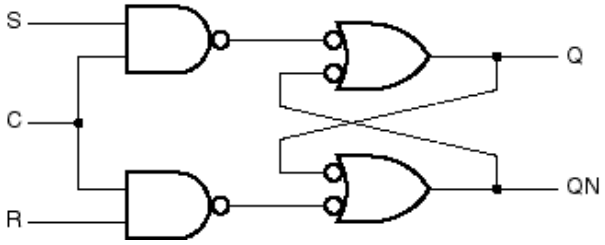


Q4. If the **input** combination **S=1, R=1** is applied to this circuit, the (steady state) output will be:

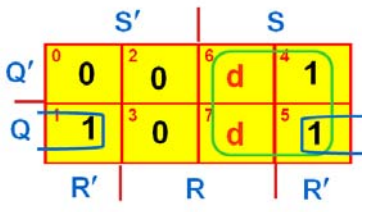
A. Q=0, QN=0
B. Q=0, QN=1
C. Q=1, QN=0
D. Q=1, QN=1
E. none of the above



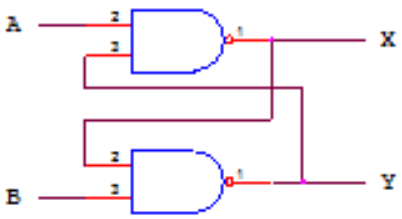
- **variations**
  - **NAND-implemented S′-R′ latch**
  - **NAND-implemented S-R latch with ENABLE (“C”)**



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



$Q^* = S + R' \cdot Q$



$X(t+\tau)$ = _____

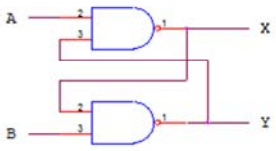
$Y(t+\tau)$ = _____

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 $X(t+\tau)$ $Y(t+\tau)$

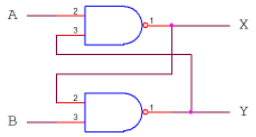
Q1. For the circuit shown, the following output combination **cannot occur at any time:**

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



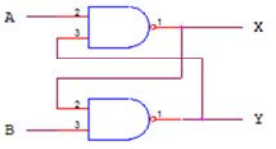
Q2. If the **input** combination **A=0, B=1** is applied to this circuit, the (steady state) output will be:

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



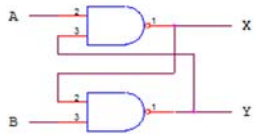
Q3. If the **input** combination **A=1, B=0** is applied to this circuit, the (steady state) output will be:

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



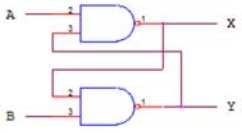
Q4. If the **input** combination **A=0, B=0** is applied to this circuit, **followed immediately** by the **input** combination **A=1, B=1,** the (steady state) output will be:

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

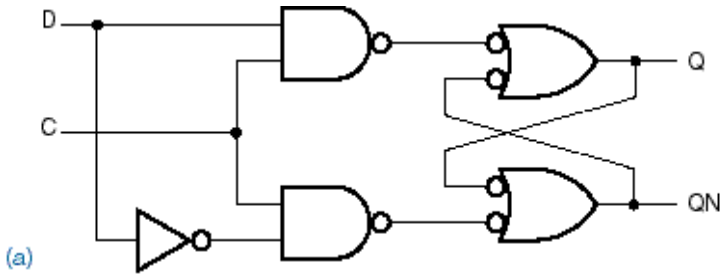


Q5. If the **propagation delay** of each gate is **10 ns,** the **minimum length of time** that (valid) input combinations need to be asserted **in order to prevent metastable behavior** is:

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

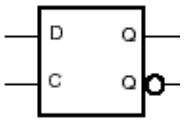


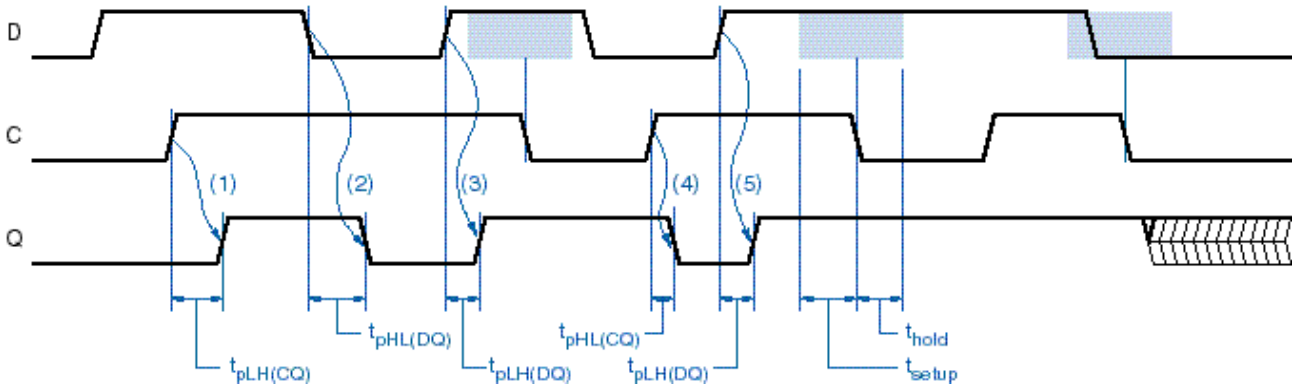
- **transparent D ("data") latch**
  - **just an S-R latch with an inverter between the S and R inputs**
  - **basic "memory bit"**
  - **called "transparent" (or "data following") because that what it is (does) when "open"**
  - **retains value when enable is negated (latch "closed")**
  - **propagation delay parameters**
  - **setup and hold times (what happens if either is violated)**



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

(b)





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

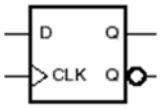
A. is always equal to its input
B. is equal to its input when the latch is closed
C. is equal to its input when the latch is open
D. changes state as soon as the latch is clocked
E. none of the above

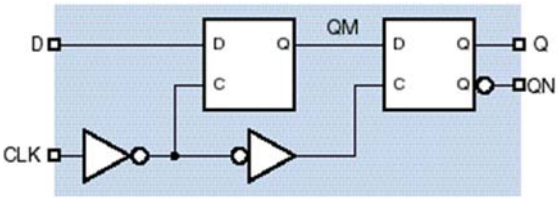
Lecture Summary – Module 3-C

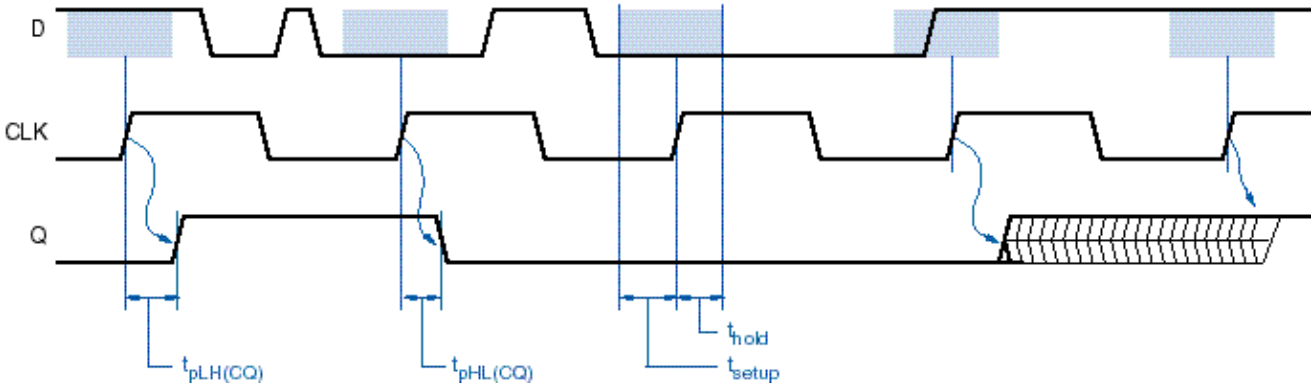
Data (D) and Toggle (T) Flip-Flops

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 532-535, 541-542

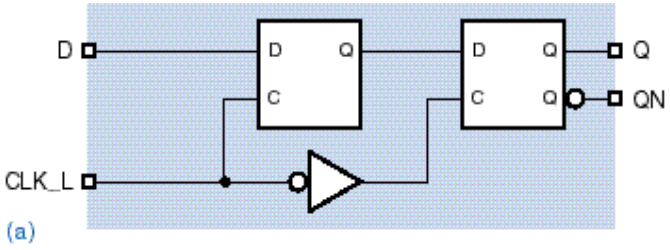
- **edge-triggered D flip-flop**
  - **changes state ("triggers") on clock edge**
  - **can be positive (rising) edge triggered or negative (falling) edge triggered**
  - **created using two latches cascaded together, that open on opposite clock phases**
    - **input latch "master"**
    - **output latch ("slave")**
  - **triangle = dynamic input indicator (clock)**
  - **characteristic equation: Q* = D**
  - **propagation delay parameters**
  - **setup and hold times**





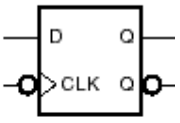


- **negative edge-triggered D flip-flop**



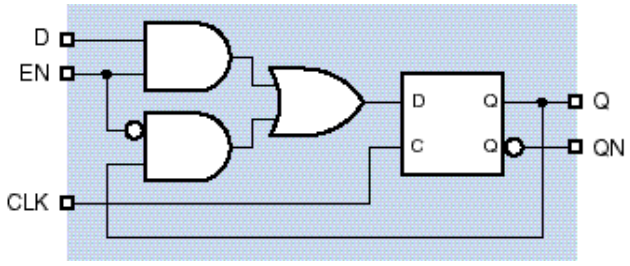
D	CLK_L	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN

(b)

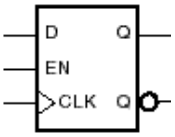


(c)

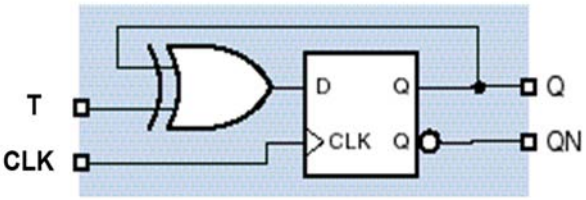
- **edge-triggered D flip-flop with enable**

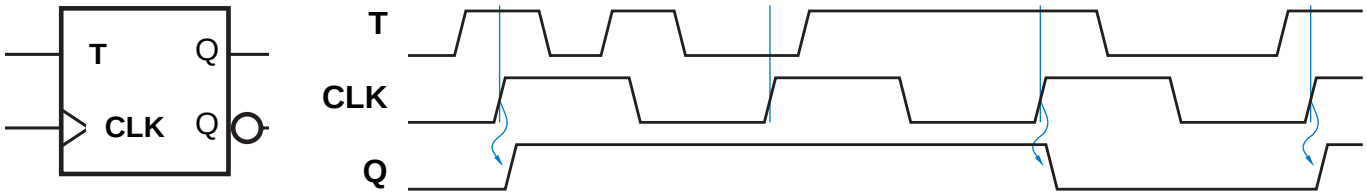


D	EN	CLK	Q	QN
0	1	↑	0	1
1	1	↑	1	0
x	0	↑	last Q	last QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN

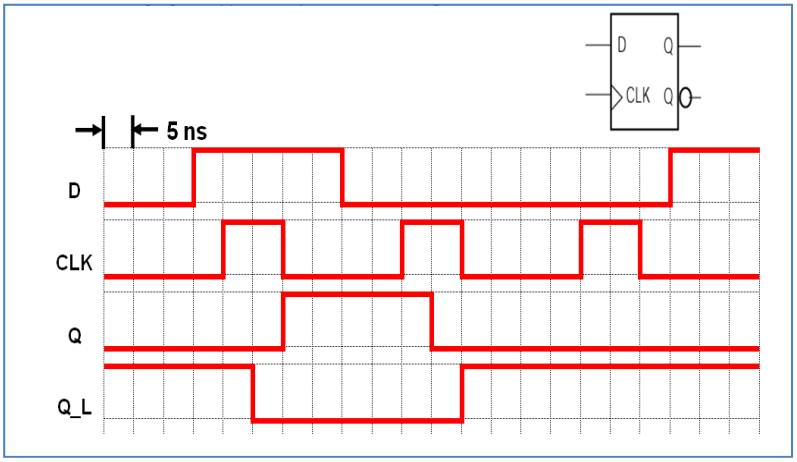


- **edge-triggered T ("toggle") flip-flop**
  - **toggles state (Q*= Q′) if T input is 1**
  - **stays in same state (Q*= Q) if T input is 0**
  - **characteristic equation: Q*= T⊕Q** ***(can synthesize using D flip-flop as "building block")***

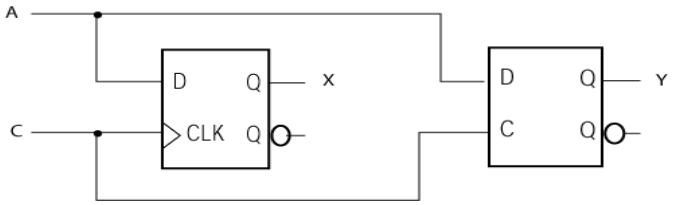


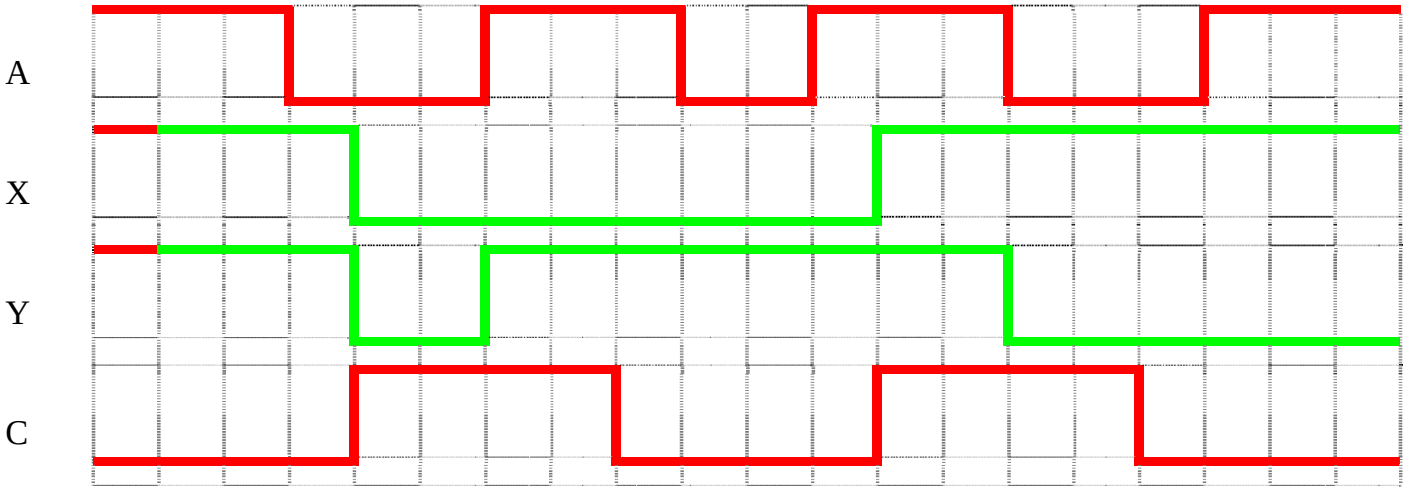


- **flip-flop timing parameters**
  - **clock pulse width**
  - **clock period**
  - **clock duty cycle**
  - **nominal setup time**
  - **nominal hold time**
  - **$t_{PLH(C\rightarrow Q)} = t_{PLH(C\rightarrow Q\_L)}$**
  - **$t_{PHL(C\rightarrow Q)} = t_{PHL(C\rightarrow Q\_L)}$**



- **response of latch vs. flip-flop**





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

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

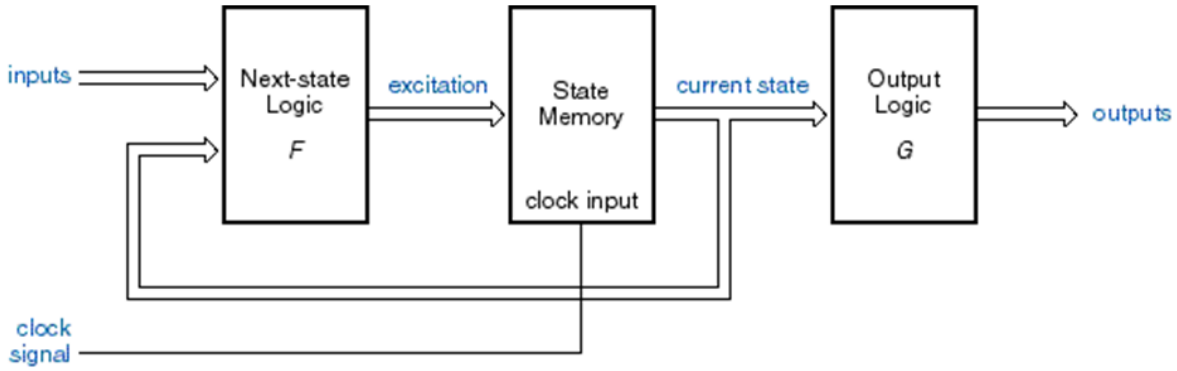
E. none of the above

Lecture Summary – Module 3-D

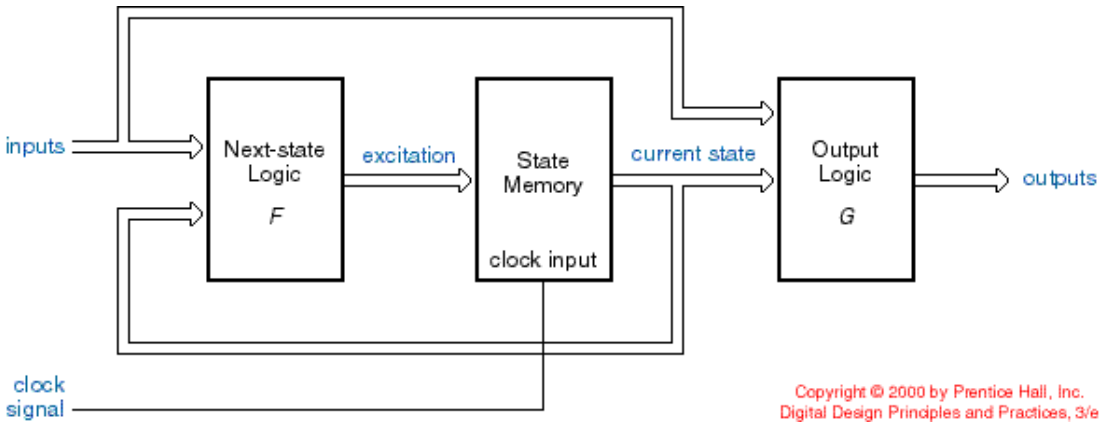
Clocked Synchronous State Machine Structure and Analysis

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 540-553

- **introduction**
  - **state machine (sequential circuit)**
  - **clocked**
  - **synchronous (all flip flops share common clocking signal)**
- **state machine basic blocks**
  - **next state ("excitation") logic**
  - **state memory (flip flops)**
  - **output logic**
- **state machine models**
  - **Moore**



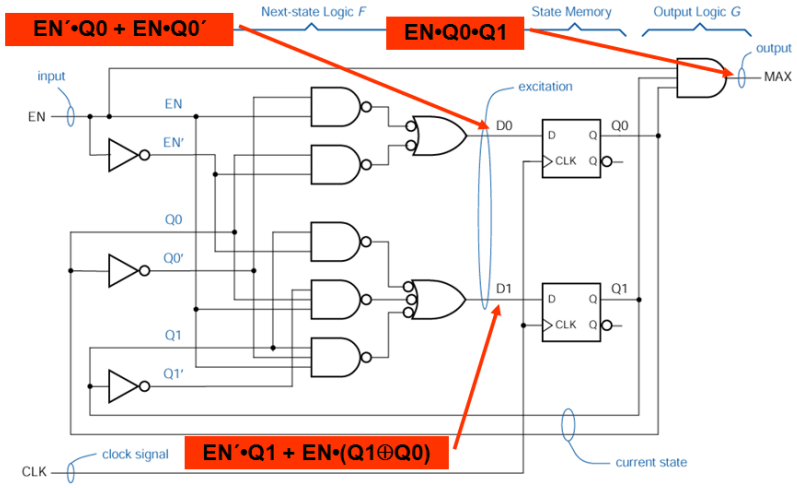
  - **Mealy**



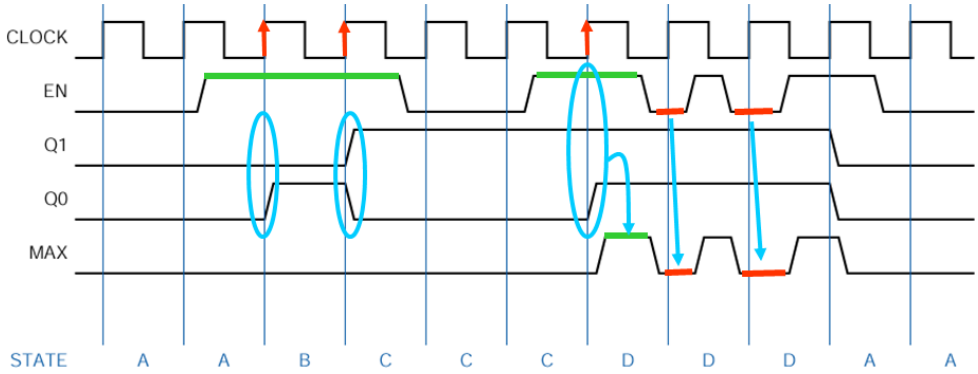
  - **can map a given state machine into either model**
  - **important: *how model chosen satisfies the design requirements***

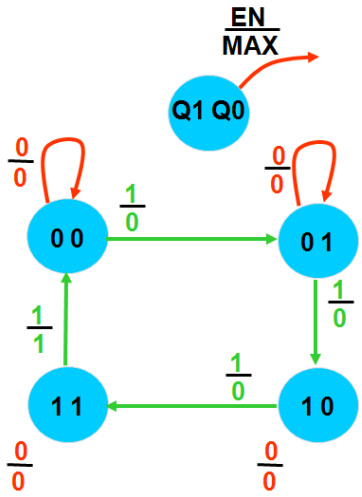
- **state machine analysis**
  - **determine next state and output functions**
  - **construct a present state – next state / output table**
  - **draw state transition diagram**
  - **draw a timing diagram**

- **example: Mealy machine analysis**

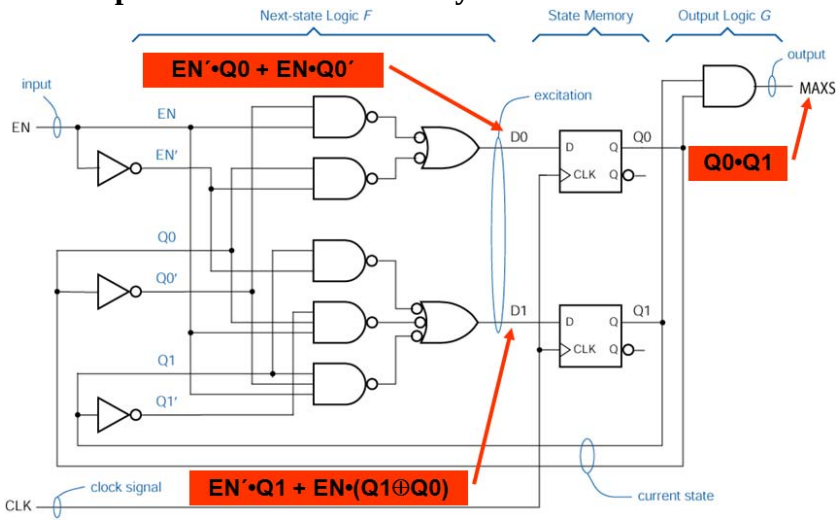


PS		PI	NS		Output
Q1	Q0	EN	Q1*	Q0*	MAX
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	0	0	1

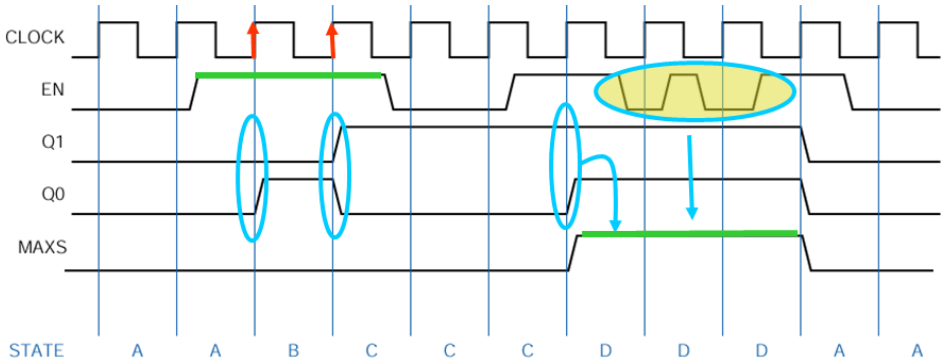


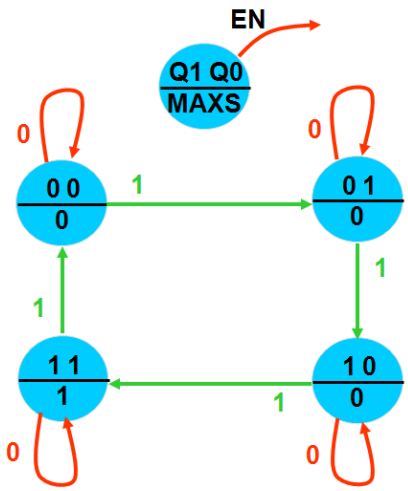


- **example: Moore machine analysis**



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





Lecture Summary – Module 3-E

Clocked Synchronous State Machine Synthesis

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 553-566, 612-625, 682-689

- **introduction – the creative process**
  - **potentially imprecise description**
  - **choose among different ways of doing things**
  - **handle special cases**
  - **keep track of several ideas in your head**
  - ***not an algorithm***
  - **circuit will perform exactly as designed**
  - **no guarantee it will work the first time**

- **state machine design steps**
  - **construct PS-NS/O table and/or STD**
  - **minimize "obvious" redundant states**
  - **assign state variable combinations**
  - **update PS-NS/O table and/or STD accordingly**
  - **(choose flip-flop type) – we will use D-type for most designs**
  - **(excitation table/equations – not needed for D-type flip flops – why?)**
  - **derive output equations**
  - **draw logic diagram or realize equations directly in a PLD (using edge-triggered D-type)**

- **derivation of 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

- **derivation of 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

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).

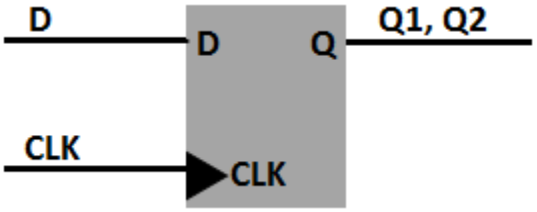
A. 1
B. 2
C. 44
D. 84
E. none of the above

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

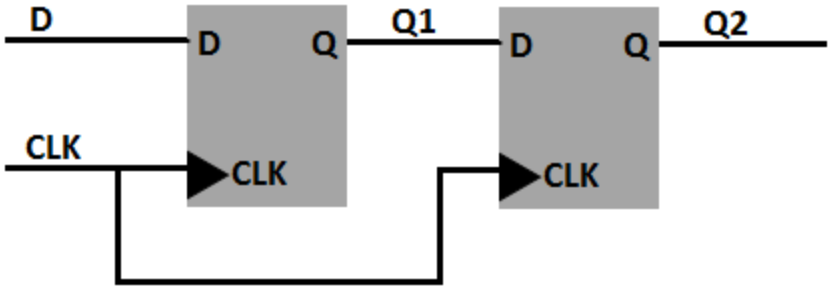
A. there are situations where *increasing* the number of states may simplify the design or reduce its cost
B. the *designer can do more* to simplify a state machine [than using formal state-minimization procedures] *during the state-assignment phase* of the design
C. by carefully matching state meanings 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**
  - **blocking statements (out = in)**
    - **the = symbol represents a *blocking* procedural assignment**
    - **used to model combinational logic**
    - **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**



  - **non-blocking statements (out <= in)**
    - **the <= symbol represents a *non-blocking* procedural assignment (analogous to a "clocked operator")**
    - **used to model sequential logic**
    - **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**

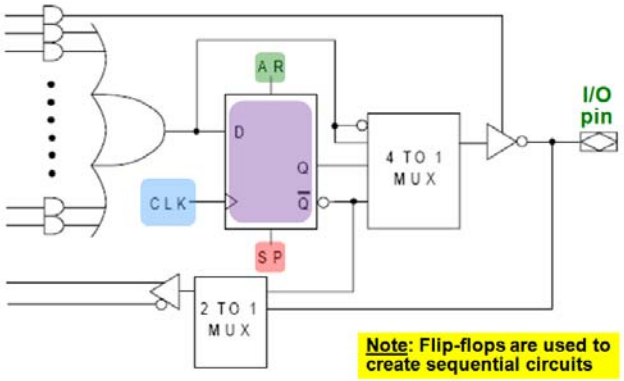


- **Verilog design guidelines**
  - **do not mix blocking and non-blocking statements in the same block or procedure**
  - **combinational blocks – use blocking statements**
  - **sequential blocks (registers) – use non-blocking statements**

- **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.**
  - **registers are assigned next-state values with non-blocking statements**
  - **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**

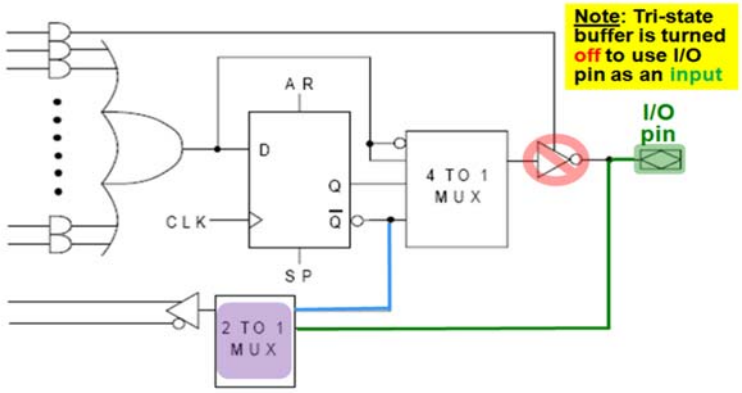
- **differences in macrocell architecture**

GAL22V10 Output Logic Macrocell ("OLMC")



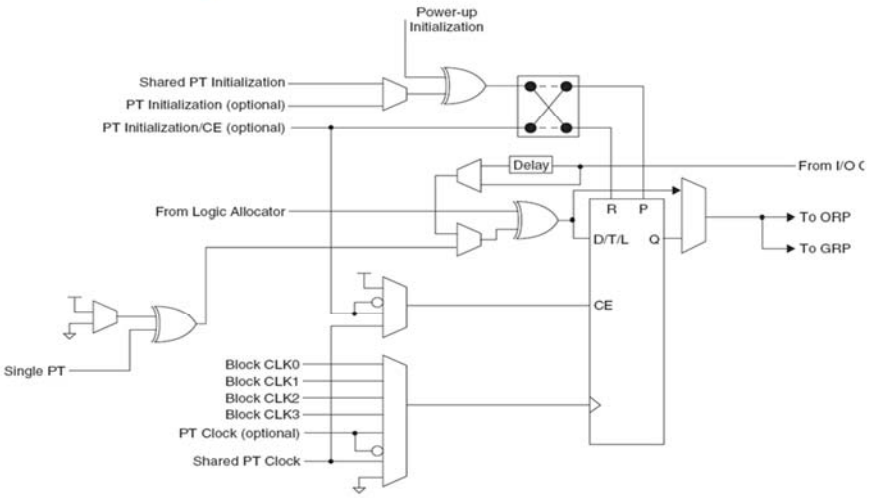
All OLMC edge-triggered D flip-flops utilize common clock (CLK), asynchronous reset (AR), and asynchronous preset (SP) signals

GAL22V10 Output Logic Macrocell ("OLMC")

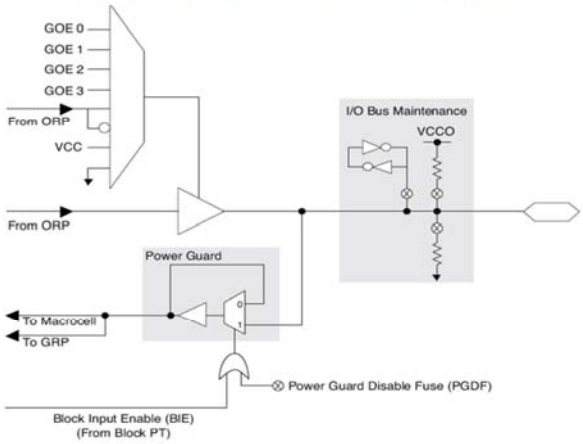


2:1 multiplexer selects (routes) true/complemented I/O pin or true/complemented registered feedback to the P-term array

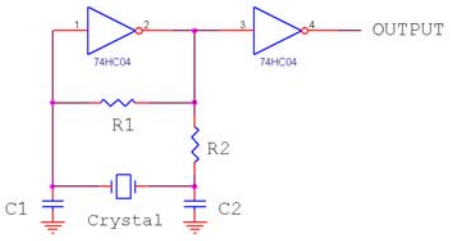
ispMACH 4000ZE Macrocell



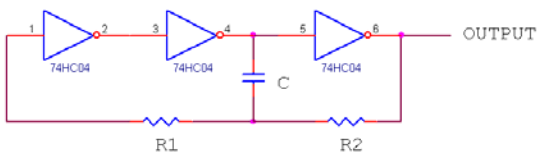
ispMACH 4000ZE I/O Cell



- **periodic clock generation circuits**
  - **typically based on crystal or R-C time constant**
  - **issues of interest**
    - **frequency**
    - **duty cycle**
    - **transition time (slew rate)**
    - **ringing (undershoot / overshoot)**
    - **stability (drift / jitter)**
    - **driving capability**
    - **skew (based on different physical path lengths)**
  - **CMOS "ring" oscillator and crystal oscillator circuits**



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

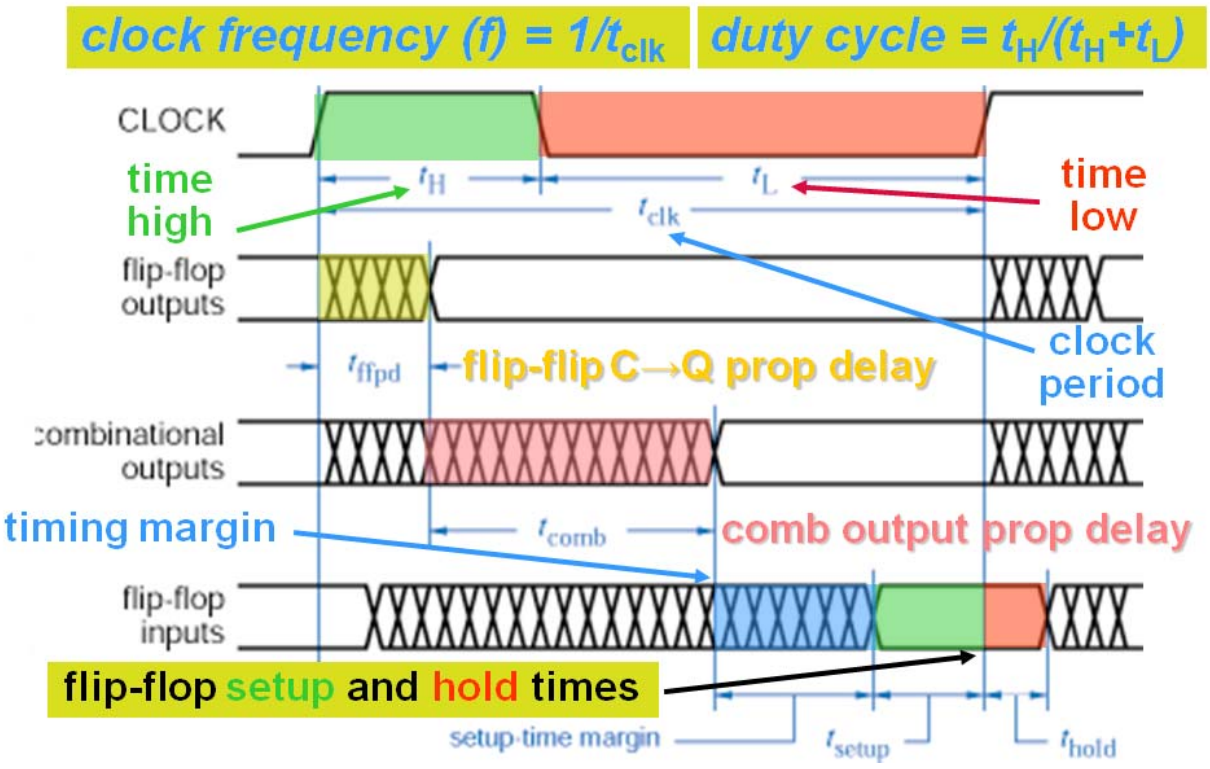


$f \cong (2C(0.4R_{eq} + 0.7R_2))^{-1}$ where $R_{eq} = (R_1R_2)/(R_1+R_2)$

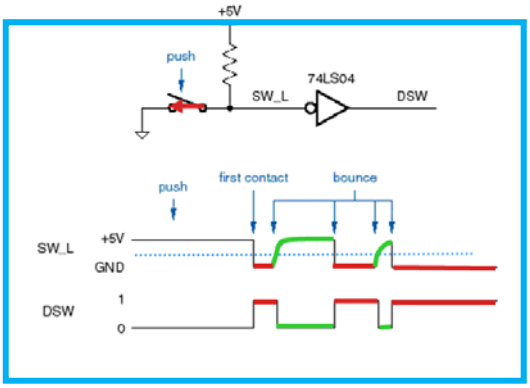
- **ispMach 4000ZE internal oscillator setup/use**

```
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
```

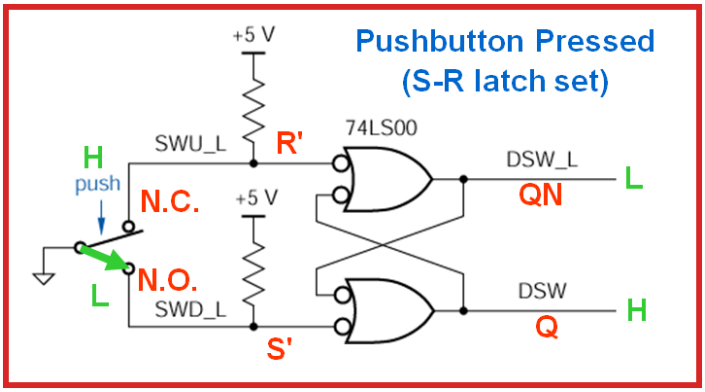
- **timing diagrams and specifications**

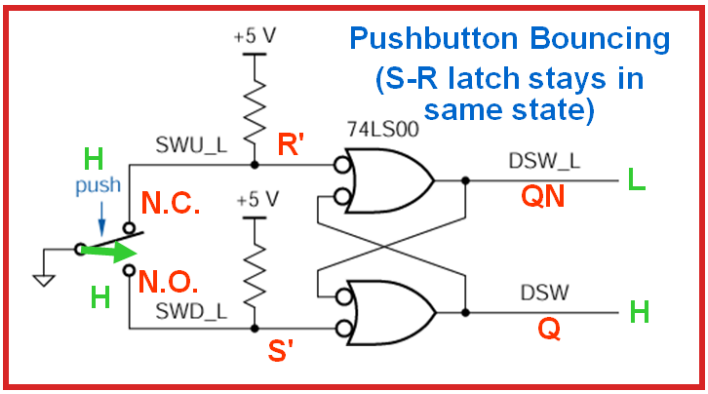


- **event clock generation circuits**
  - **examples of events**
    - **pushing button**
    - **sensor firing**
  - **problem: contact bounce**



Solution: "bounce-free" (or "bounce-less") switch implemented using a S.P.D.T. (single pole, double throw pushbutton and an S′R′ latch






```
/* 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

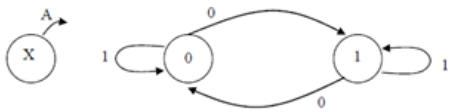
```
/* D flip flop used as bounce-free switch in Verilog */
module DFF_BF(CLK, AR, AP, D, BFC);
  input wire CLK;   // Clock input for DFF
  input wire AR,AP; // Asynchronous Reset and Preset
  input wire D;     // Data input for DFF
  output reg BFC;   // Bounce Free Switch output
always @ (posedge 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

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

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:

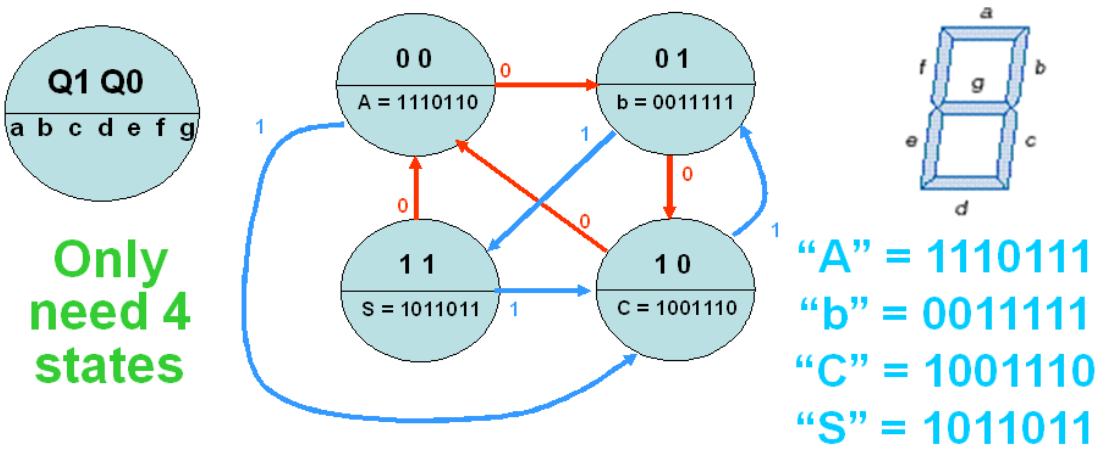
A. $X^* = A' \cdot X' + A \cdot X$
B. $X^* = A' \cdot X + A \cdot X'$
C. $X^* = A + X$
D. $X^* = A \cdot X$
E. none of the above

Lecture Summary – Module 3-F

State Machine Design Examples: Sequence Generators

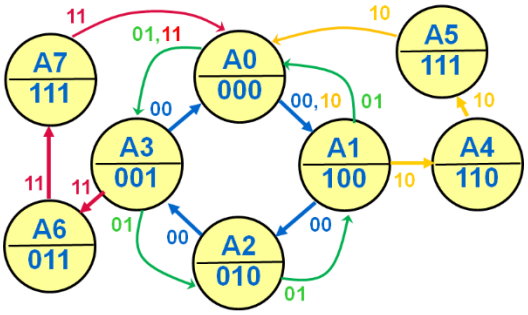
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 566-576

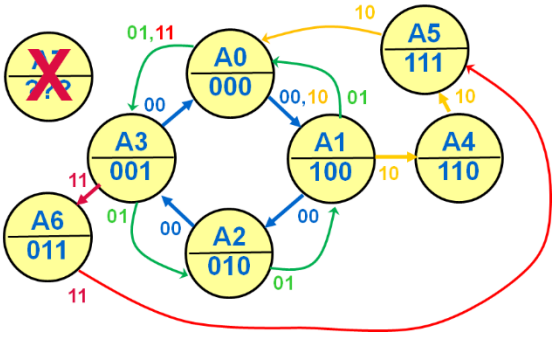
- **a *sequence generator* state machine produces a (periodic) *series of output signal assertions* that constitute a *pre-defined pattern***
- **two different design strategies**
  - **minimum cost (*don't cares* in next states are allowed)**
  - **minimum risk (unused states explicitly assigned a next state)**
- **character sequence display – displays AbC or CbS on a 7-segment display (Moore model)**



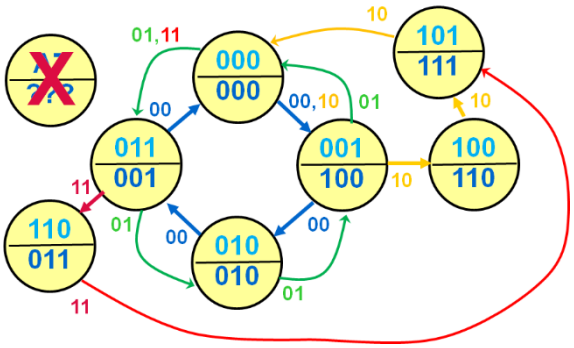

```
module tv_disp(CLK, M, Q, nL);
  input wire CLK;
  input wire M;   // Mode control
  output reg [1:0] Q;
  output wire [6:0] nL;
  reg [6:0] L;    // L[6] = LA, L[5] = LB, .. L[0] = LG
  reg [1:0] next_Q;
  assign nL = ~L; // Active-low outputs on L
  always @ (posedge CLK) begin
    Q <= next_Q;
  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

    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
```

- **4-mode light sequencer – Moore model**





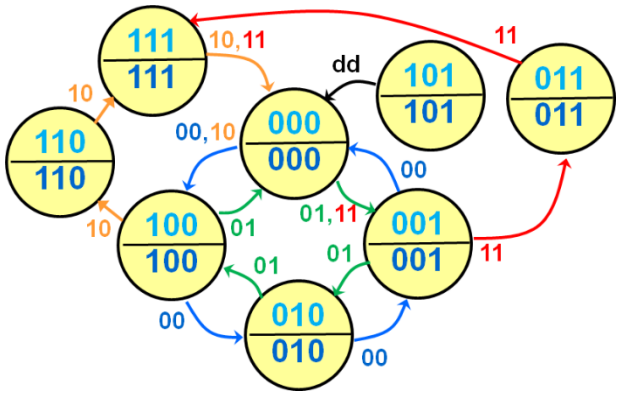
PS Q2	PS Q1	PS Q0	PI M1	PI M0	NS Q2*	NS Q1*	NS Q0*	PO L2	PO L1	PO L0
0	0	0	0	0	0	0	1	0	0	0
			0	1	0	1	1			
			1	0	0	0	1			
			1	1	0	1	1			
0	0	1	0	0	0	1	0	1	0	0
			0	1	0	0	0			
			1	0	1	0	0			
			1	1	0	0	0			
0	1	0	0	0	0	1	1	0	1	0
			0	1	0	0	1			
			1	0	0	0	0			
			1	1	0	0	0			
0	1	1	0	0	0	0	0	0	0	1
			0	1	0	1	0			
			1	0	0	0	0			
			1	1	1	1	0			
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			




```
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;
      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

- **check alternate state/output assignments (where output functions are the state variables)**

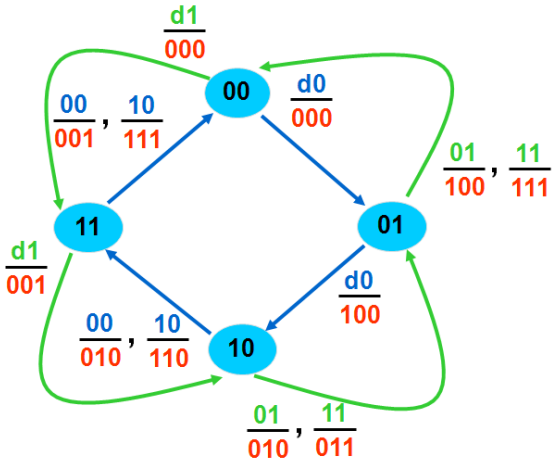



```
module moorelsB(CLK, M, Q);
  input wire CLK;     // Input clock
  input wire [1:0] M; // Mode select
  output reg [2:0] Q; // serve as L2 L1 L0
  reg [2:0] next_Q;
  always @ (posedge CLK) begin
    Q <= next_Q;
  end

    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;
        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
```

```
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 decalarations
  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
      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
```

Both realizations (clocked operator table and state diagram) use 3 macrocells

- **Mealy model**



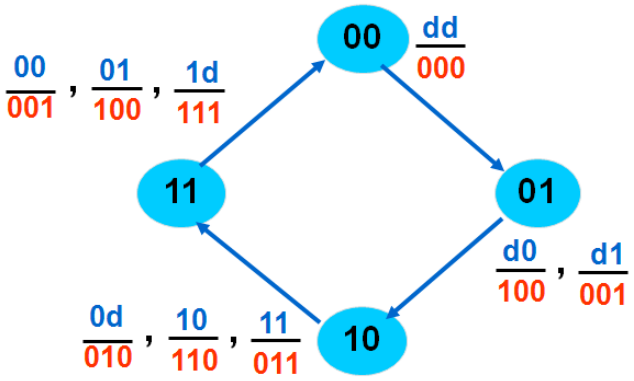
PS Q1	PS Q0	PI M1	PI M0	NS Q1*	NS Q0*	PO L2	PO L1	PO L0
0	0	0	0	0	1	0	0	0
		0	1	1	1	0	0	0
		1	0	0	1	0	0	0
		1	1	1	1	0	0	0
0	1	0	0	1	0	1	0	0
		0	1	0	0	1	0	0
		1	0	1	0	1	0	0
		1	1	0	0	1	1	1
1	0	0	0	1	1	0	1	0
		0	1	0	1	0	1	0
		1	0	1	1	1	1	0
		1	1	0	1	0	1	1
1	1	0	0	0	0	0	0	1
		0	1	1	0	0	0	1
		1	0	0	0	1	1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

```
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

- **check alternate Mealy state/output assignments**

```
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;

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

  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
```




```
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
  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
```

Both realizations (clocked operator table and state diagram) use 5 macrocells

- **conclusions**
  - **choosing the "right" state variable assignment and machine model can make a significant difference in the PLD resources consumed and the amount of work required**
  - **the only formal way to find the "best" assignment is to try *all* of the assignments**
  - **experience is needed to do this well (see text for guidelines)**
  - **there is no substitute for practice (developing "applied intuition")**

```
/* 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
```

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

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

Lecture Summary – Module 3-G

State Machine Design Examples: Counters and Shift Registers

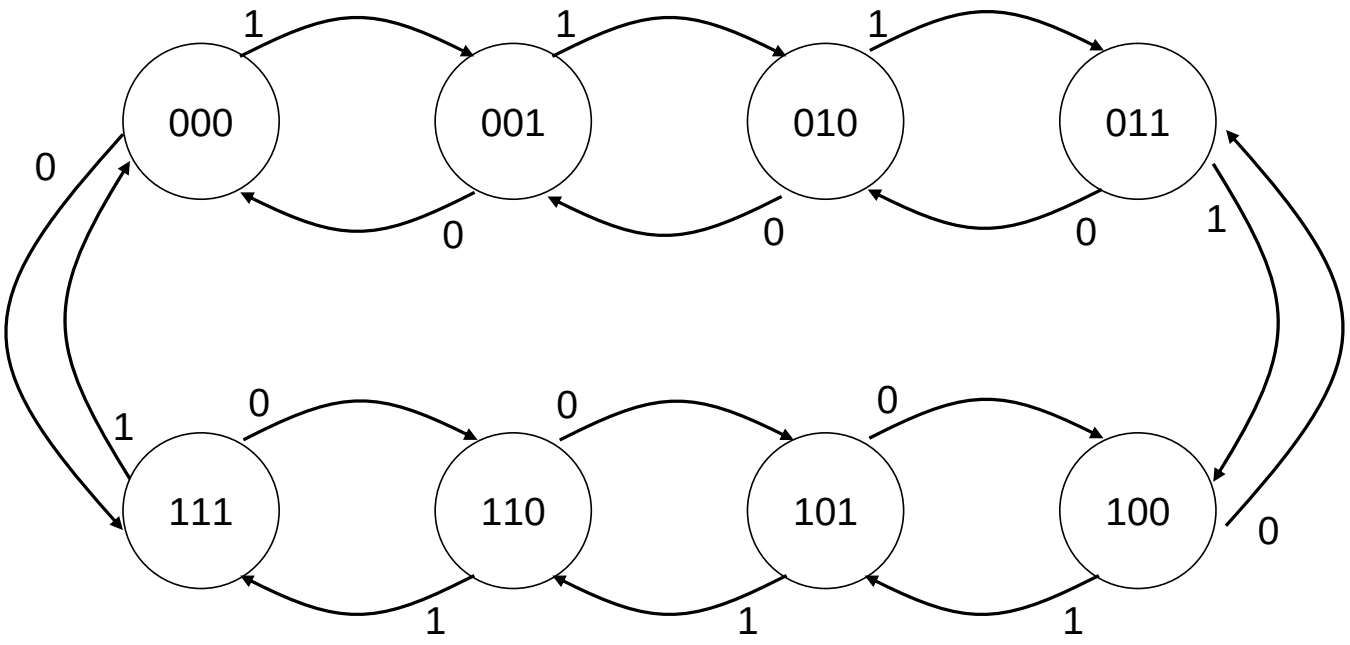
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 710-721, 727-736

- **the term *counter* is used for any clocked sequential circuit whose state diagram contains a *single cycle***
  - **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*)**
  - **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***
  - **UP counter K^{th} bit next state: $Q_K^* = Q_K \oplus (Q_{K-1} \bullet Q_{K-2} \bullet \ldots \bullet Q_1 \bullet Q_0)$**
  - **DOWN counter K^{th} bit next state: $Q_K^* = Q_K \oplus (Q'_{K-1} \bullet Q'_{K-2} \bullet \ldots \bullet Q'_1 \bullet Q'_0)$**
  - **Verilog program for 8-bit UP/DOWN 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
```

  - **Verilog program for 8-bit resettable 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
```



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 (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
```

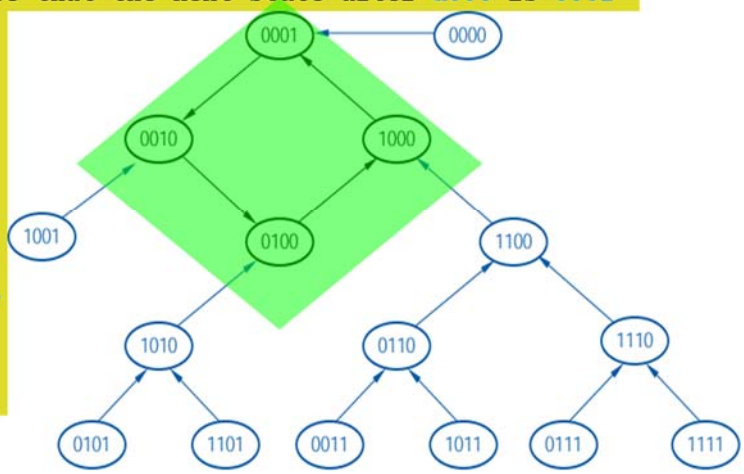
```
/* Program (B) */
module CQ(CLK, M, Q);
  input wire CLK, M;
  output reg [2:0] Q;
  reg [2:0] next_Q;
  always @ (posedge CLK) begin
    Q <= next_Q;
  end
  always @ (Q, M) begin
   next_Q[0] = ~Q[0];
   next_Q[1] =  Q[1] ^ (~M&Q[0] | M&~Q[0]);
   next_Q[2] =  Q[2] ^ (~M& Q[1]& Q[0] |
                M&~Q[1]&~Q[0]);

  end
endmodule
```

```
/* Program (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

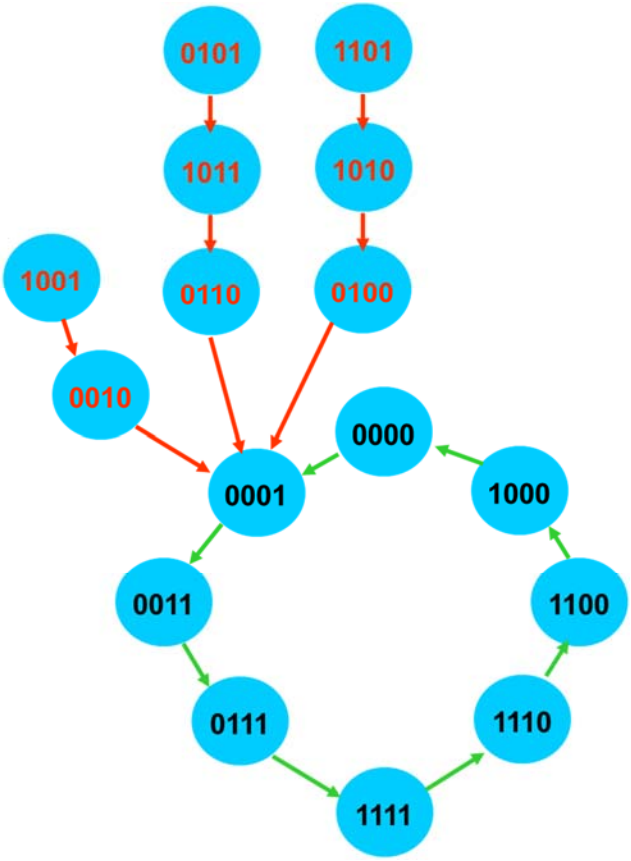
- **a shift register whose state diagram is *cyclic* is called a *shift-register counter* (i.e., does not count "up" or "down")**
  - **self-correcting ring counter**

```
/* 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 d000 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
```

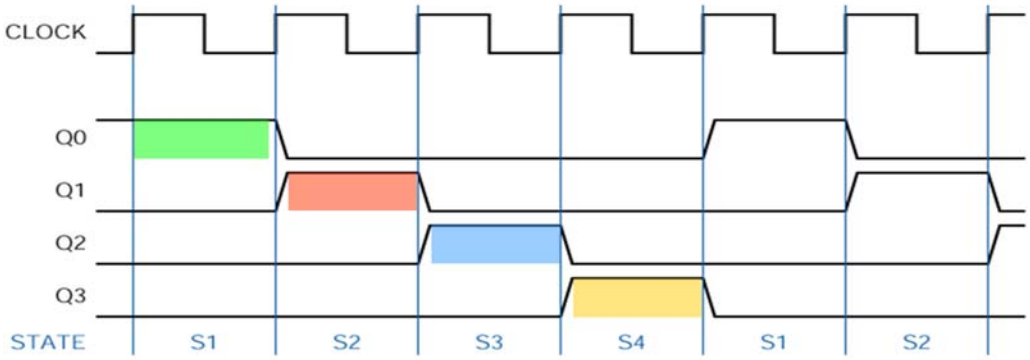


  - **self-correcting Johnson counter**

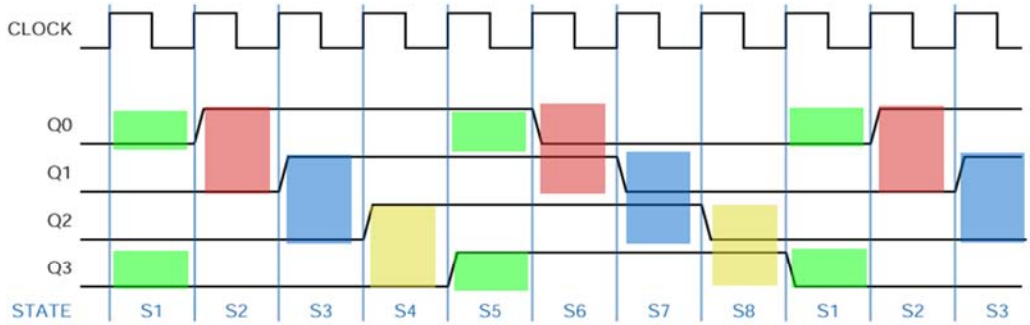
```
module john4sc(CLK, Q);

  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**
  - **ring – *none* (“one hot”), glitch-free**



  - **Johnson – 2n two-input AND or NAND gates, glitch-free**



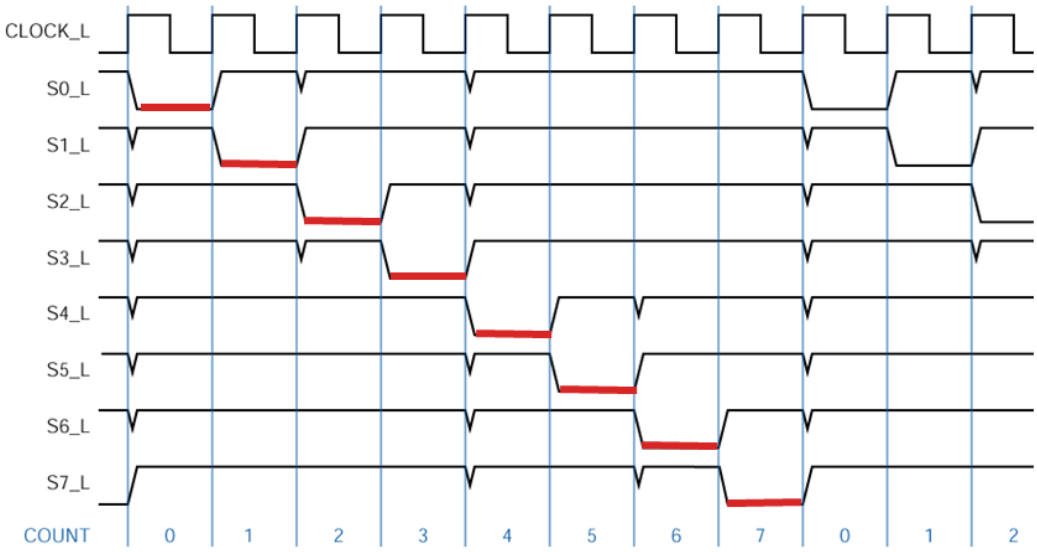
$S1 = Q0' \cdot Q3'$ $S5 = Q0 \cdot Q3$

$S2 = Q0 \cdot Q1'$ $S6 = Q0' \cdot Q1$

$S3 = Q1 \cdot Q2'$ $S7 = Q1' \cdot Q2$

$S4 = Q2 \cdot Q3'$ $S8 = Q2' \cdot Q3$

  - **comparison with binary counter state decoding – *not* glitch-free**



  - **n-bit counter with 2^n states that <u>can</u> be decoded glitch-free: Gray-code**

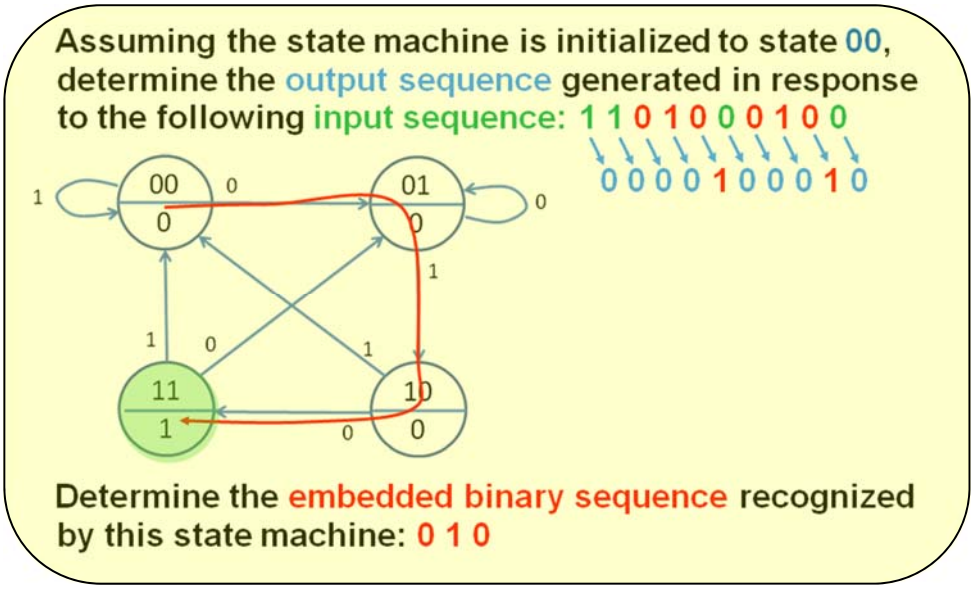
Lecture Summary – Module 3-H

State Machine Design Examples: Sequence Recognizers

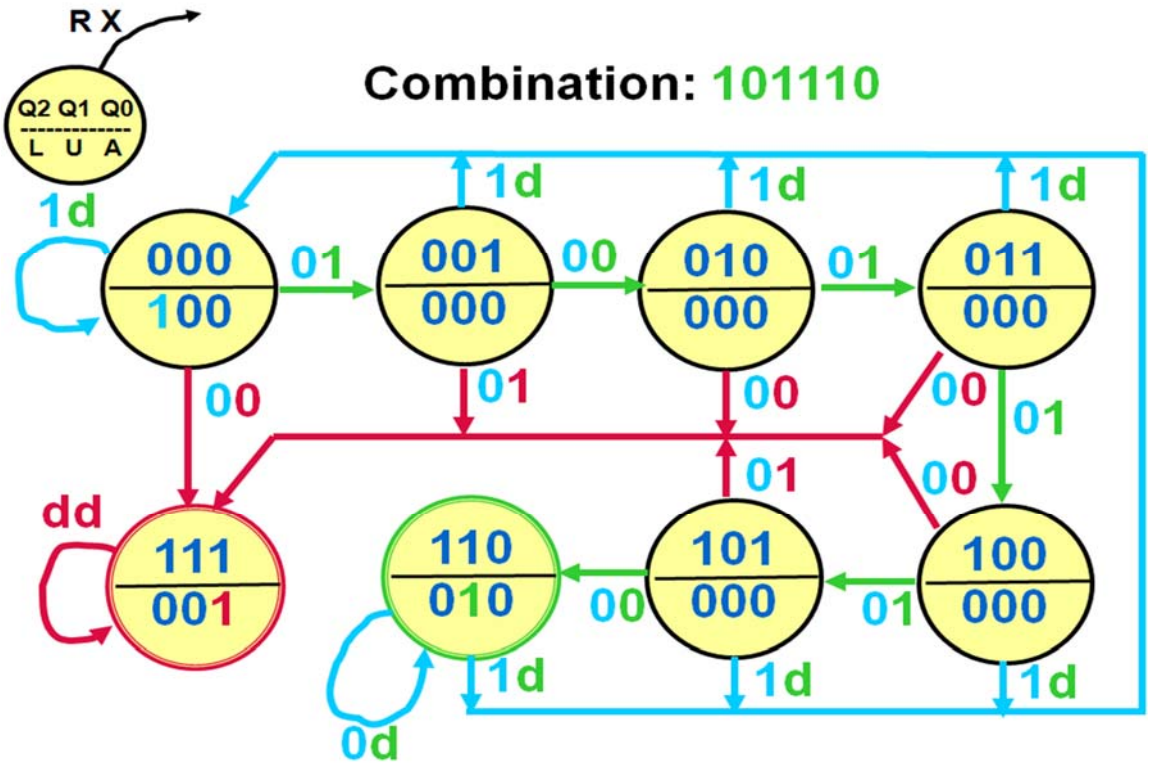
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 580-587

- **a *sequence recognizer* state machine responds to a *pre-defined input pattern* of signal assertions and produces corresponding output signal assertions**
- **use of Moore model generally preferred**
- **special states**
  - **final state of accepting sequence (pattern being recognized)**
  - **trap state**

- **simple embedded sequence recognizer**



- **digital combination lock**
  - **fixed (“hard wired”) combination**
  - **three input signals**
    - **X – combination data**
    - **R – (synchronous) relock**
    - **RESET – asynchronous reset (only way out of trap state)**
  - **three output signals**
    - **LOCKED**
    - **UNLOCKED**
    - **ALARM**
  - **Moore model**
    - **(initial) “locked” state**
    - **six states to accept combo**
    - **“alarm” state**
    - **total states needed: 8**
  - **types of states**
    - **accepting sequence (entering combination)**
    - **final state (sequence correctly entered)**
    - **trap state (error made while entering combination)**




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

  input wire CLK, RST, X, R;
  // X = lock 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

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

```
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
endmodule
```