System I

Sequential Logic Design

Haifeng Liu

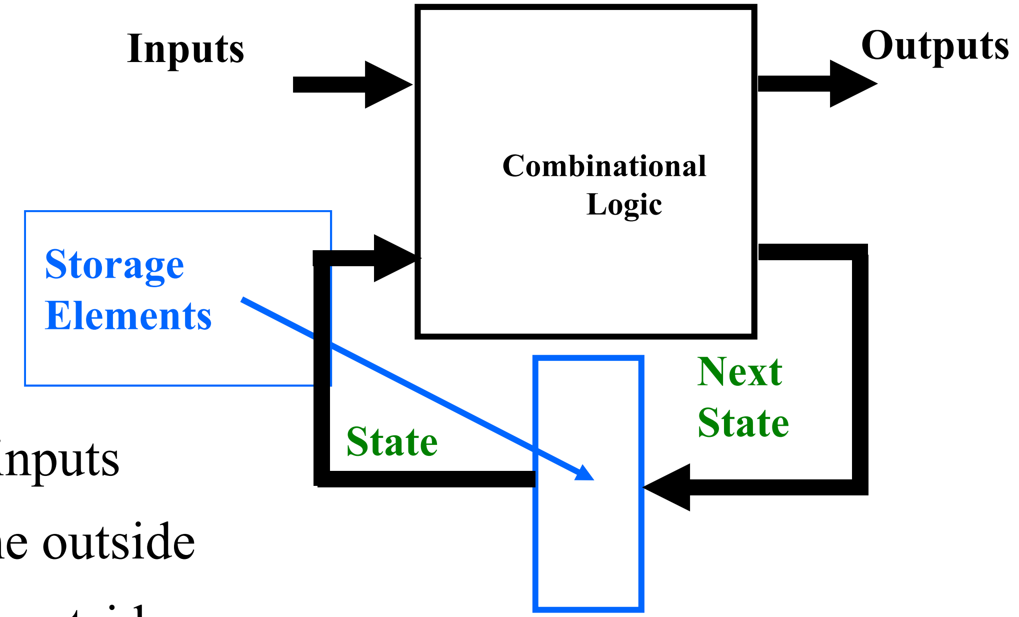
Zhejiang University

Overview

- Introduction of sequential circuits
- Basic storage elements
- Synchronous sequential logic analysis
- Synchronous sequential logic design
- Classic sequential logic circuits

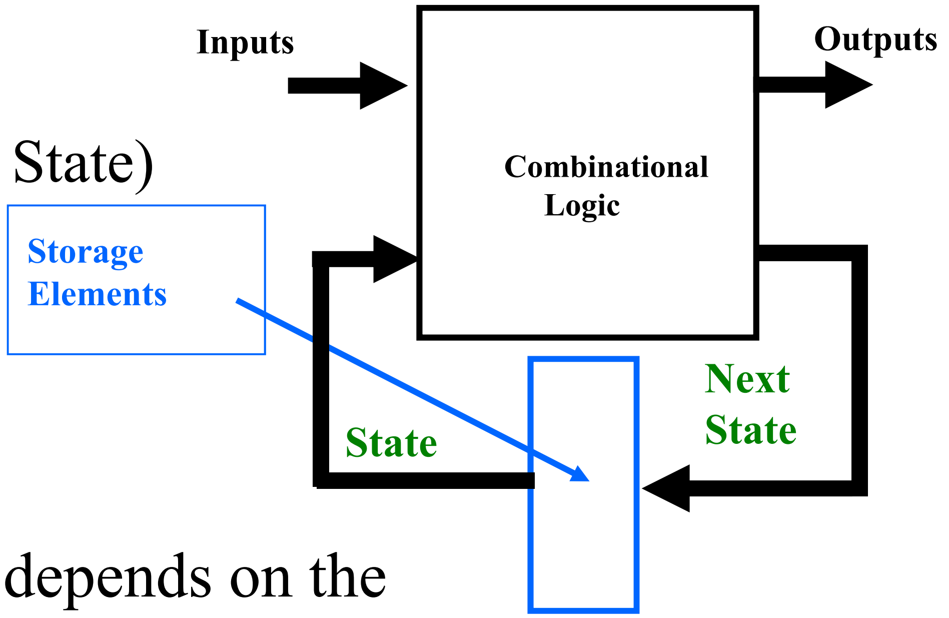
Introduction to Sequential Circuit

- Consist of:
  - Storage elements:
    Latches or flip-flops
  - Combinational Logic:
    - Implement a function of inputs
    - Inputs are signals from the outside
    - Outputs are signals to the outside
    - Other Signals: Present State, signal from the storage elements
    - Other output, Next state: input of storage element



Introduction to Sequential Circuit

- Combinational Logic
  - *Function of next state*
    Next State= f(Inputs, Present State)
  - *Output Function*(Mealy)
    Output = g(Inputs, State)
  - *Output Function*(Moore)
    Output = h (Present State)
  - The type of output function depends on the functionality, and have a great impact on the design



Types of Sequential Circuits

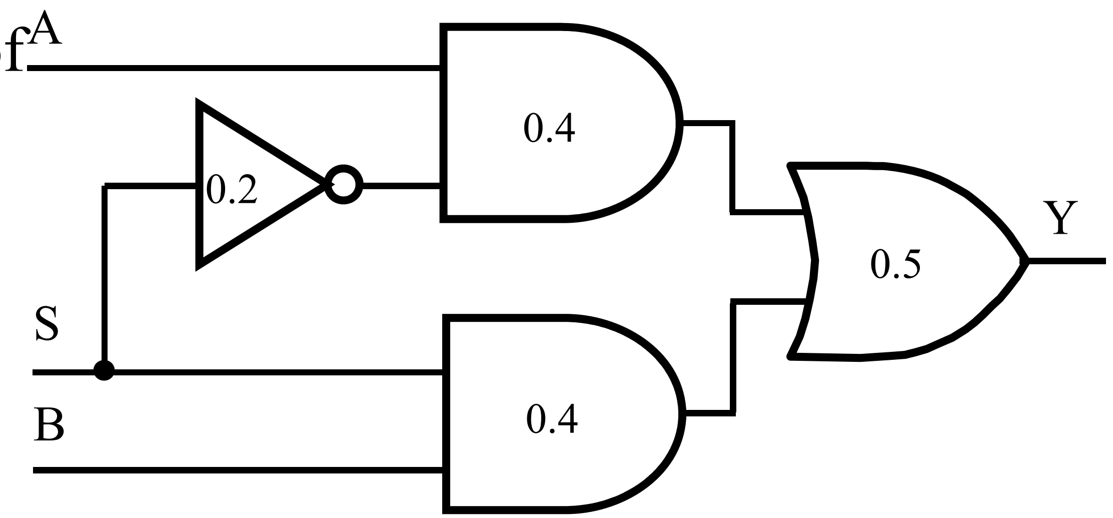
- Depends on the times at which:
  - storage elements observe their inputs, and
  - storage elements change their state
- Synchronous
  - Behavior defined by knowledge of its signals at discrete instances of time
  - Storage elements observe inputs and can change state only in relation to a timing signal (clock pulses from a clock)
- Asynchronous
  - Behavior defined by knowledge of inputs at any instant of time and the order in continuous time in which inputs change
  - If clock is regarded as another input, all circuits are asynchronous!
  - Nevertheless, the synchronous abstraction makes complex designs tractable!

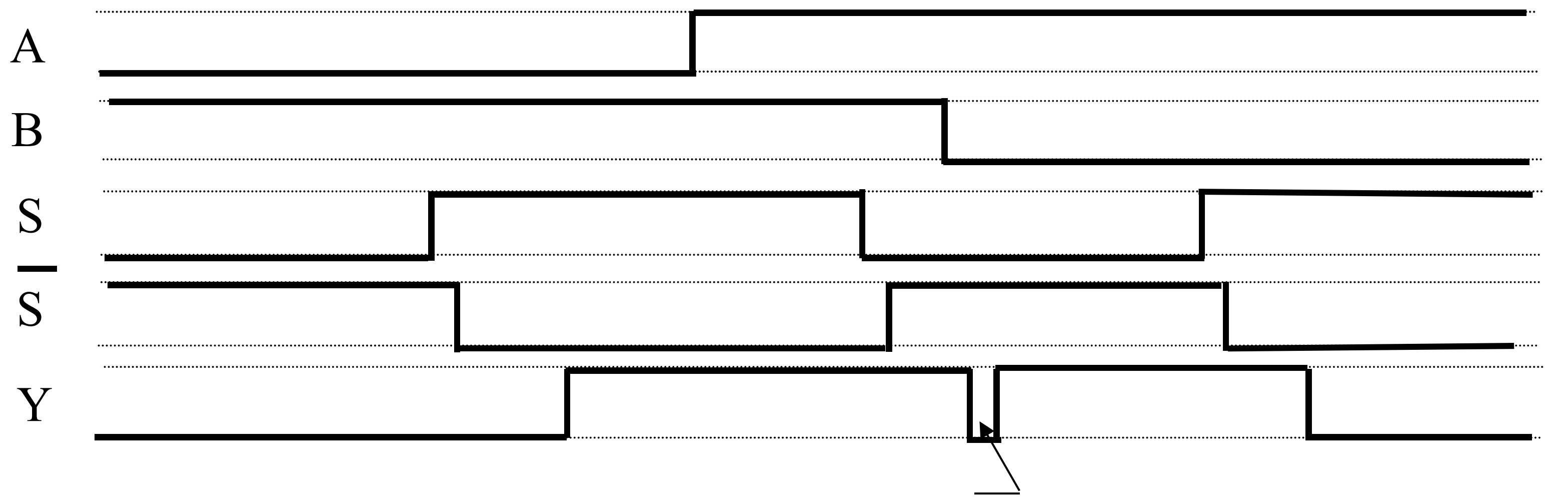
Overview

- Introduction to sequential circuits
- Basic storage elements
- Synchronous sequential logic analysis
- Synchronous sequential logic design
- Classic sequential logic circuits

Gate Delay Models

- Consider a simple of 2-input multiplexer with function:
  - Y = A for S = 0
  - Y = B for S = 1

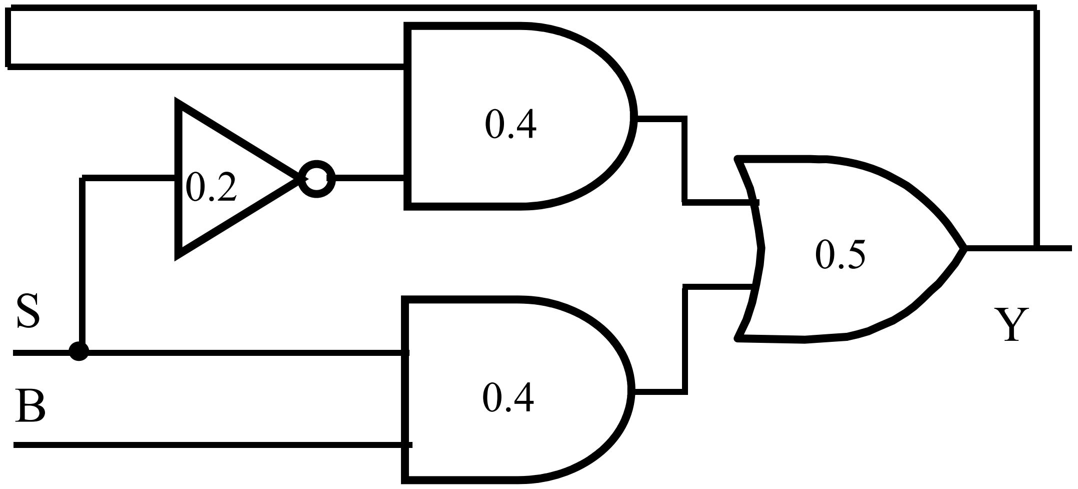


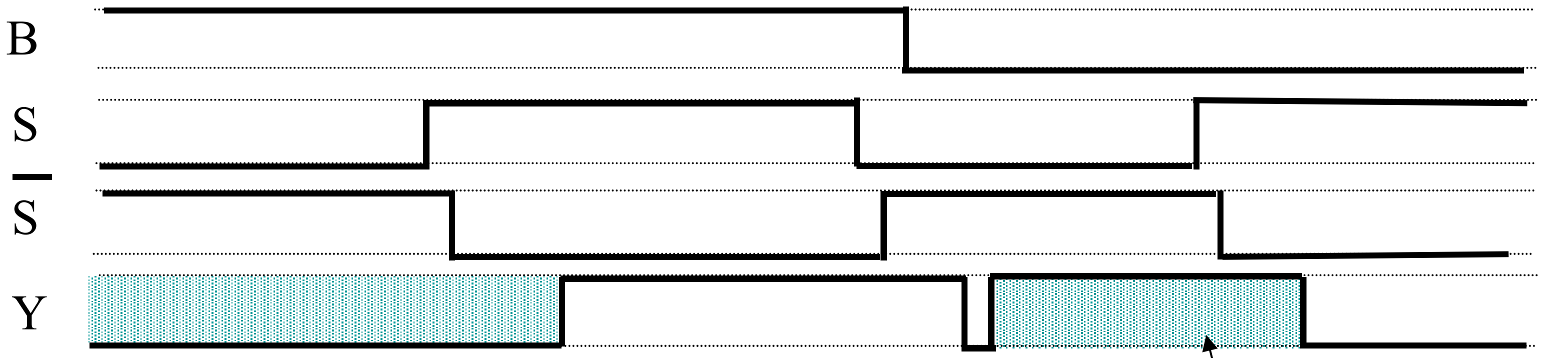


"Glitch" is due to delay of inverter

Circuit Delay Model

- What if Y connected to A?
- Circuit becomes:
- With function:
  - Y = B for S = 1, and Y(t) dependent on Y(t – 0.9) for S = 0





- The simple combinational circuit now becomes a sequential circuit because its output is a function of a time sequence of input signals!

Y is stored value in shaded area

Storing State (Continued)

- Simulation example as input signals change with time. Changes occur every 100 ns, so that the 1 ns delays are negligible.

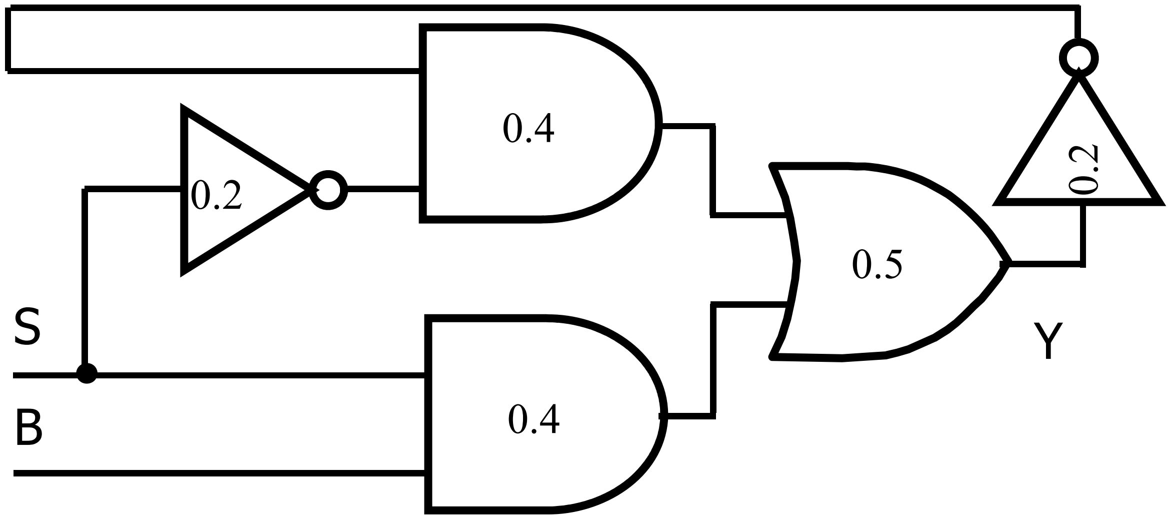
Time ↓

B	S	Y	Comment
1	0	0	Y “remembers” 0
1	1	1	Y = B when S = 1
1	0	1	Now Y “remembers” B = 1 for S = 0
0	0	1	No change in Y when B changes
0	1	0	Y = B when S = 1
0	0	0	Y “remembers” B = 0 for S = 0
1	0	0	No change in Y when B changes

- Y represents the <u>state</u> of the circuit, not only an output.

Storing State (Continued)

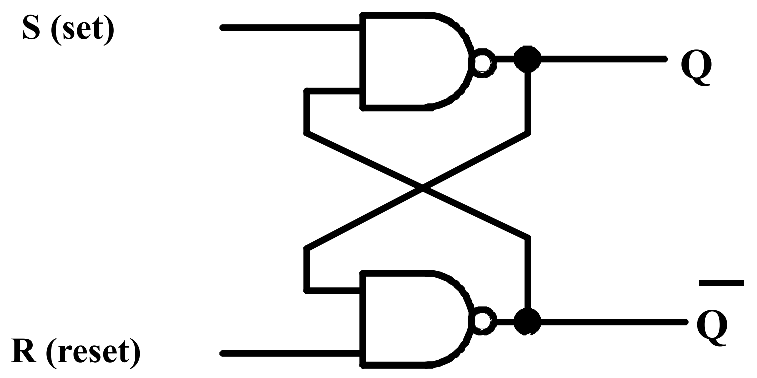
- Suppose we place an inverter in the "feedback path"
- The following are behavior results:
- The circuit is said to be unstable.
- For S = 0, the circuit becomes what is called an *oscillator*. Can be used as crude clock.



B	S	Y	Comment
0	1	0	Y = B when S = 1
1	1	1	
1	0	1	Now Y "remembers" 1
1	0	0	Y changes after 1.1 ns
1	0	1	Y changes after 1.1 ns
1	0	0	Y changes after 1.1 ns

Basic (NAND) $\overline{S} – \overline{R}$ Latch

- "Cross-Coupling" two NAND gates gives the $\overline{S}$ -$\overline{R}$ Latch:
- Which has the time sequence behavior:



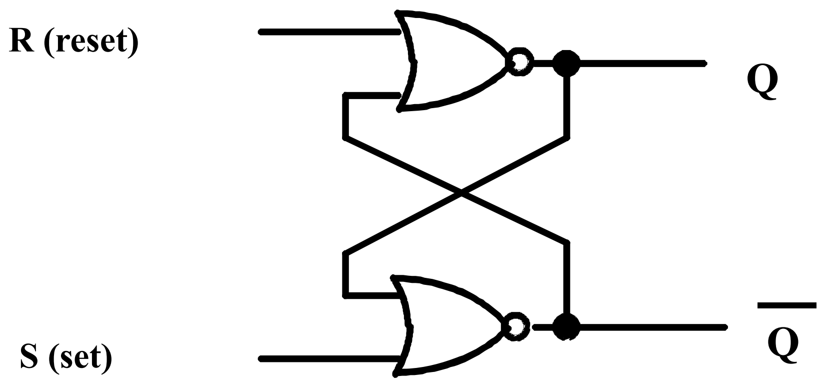
Time ↓

R	S	Q	$\overline{Q}$	Comment
1	1	?	?	Stored state unknown
1	0	1	0	"Set" Q to 1
1	1	1	0	Now Q "remembers" 1
0	1	0	1	"Reset" Q to 0
1	1	0	1	Now Q "remembers" 0
0	0	1	1	Both go high
1	1	?	?	Unstable!

- S = 0, R = 0 is <u>forbidden</u> as input pattern

Basic (NOR) S – R Latch

- Cross-coupling two NOR gates gives the S – R Latch:
- Which has the time sequence behavior:

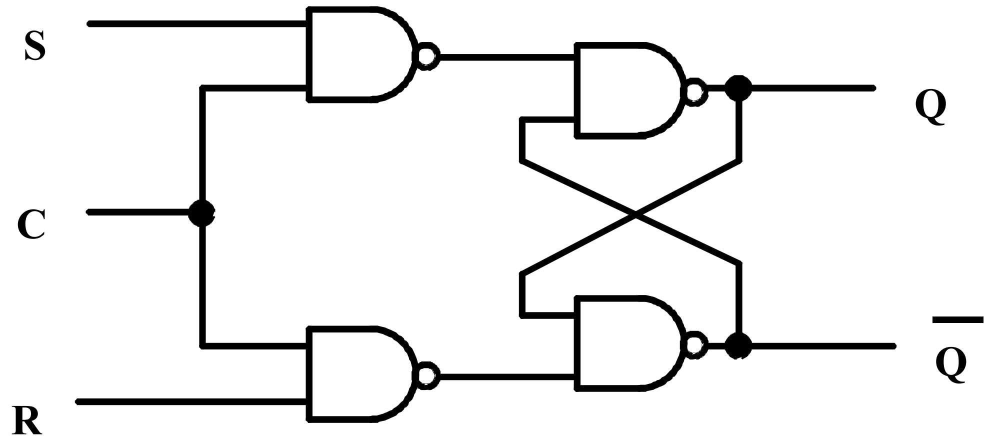


Time ↓

R	S	Q	$\overline{Q}$	Comment
0	0	?	?	Stored state unknown
0	1	1	0	"Set" Q to 1
0	0	1	0	Now Q "remembers" 1
1	0	0	1	"Reset" Q to 0
0	0	0	1	Now Q "remembers" 0
1	1	0	0	Both go low
0	0	?	?	Unstable!

Clocked S - R Latch

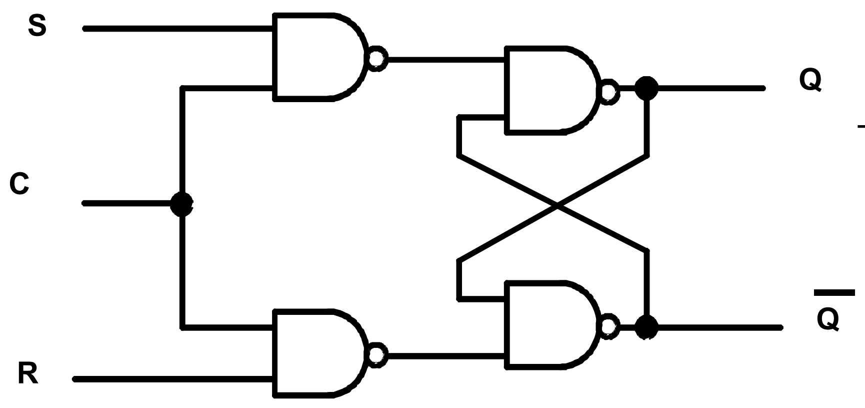
Adding two NAND gates to the basic $\overline{S}$ - $\overline{R}$ NAND latch gives the clocked S – R latch:



- Has a time sequence behavior similar to the basic S-R latch <u>except that</u> the S and R inputs are only observed when the line C is high.
- C means “control” or “clock”.

Clocked S - R Latch (continued)

- The Clocked S-R Latch can be described by a table:

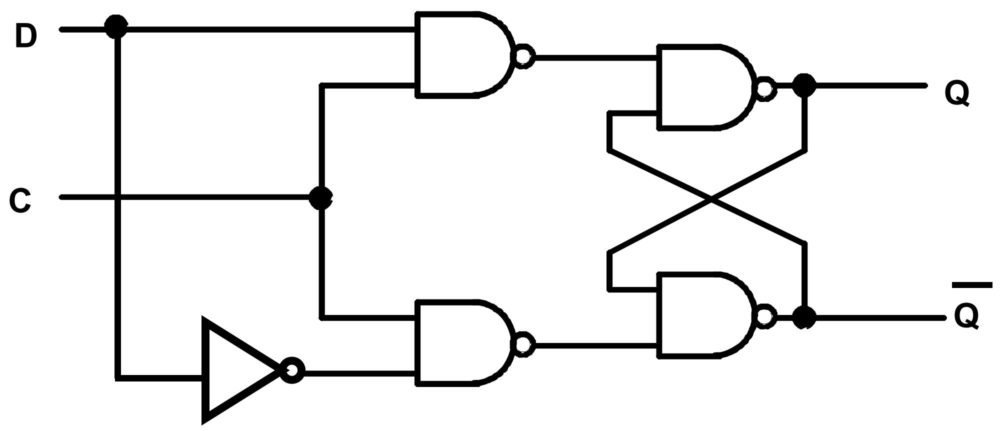


C	S	R	Q(t + 1)
0	X	X	No change
1	0	0	No change
1	0	1	Q=0: Clear Q
1	1	0	Q=1: Set Q
1	1	1	Undefined

- The table describes what happens after the clock [at time (t+1)] based on:
  - current inputs (S, R, C)

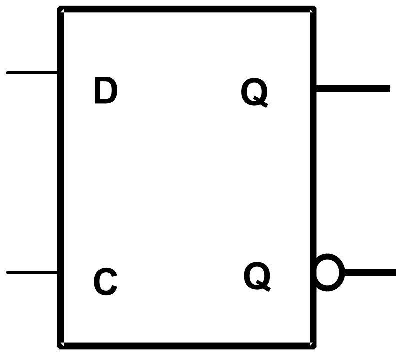
D Latch

- Adding an inverter to the S-R Latch, gives the D Latch:
- Note that there are no "indeterminate" states!

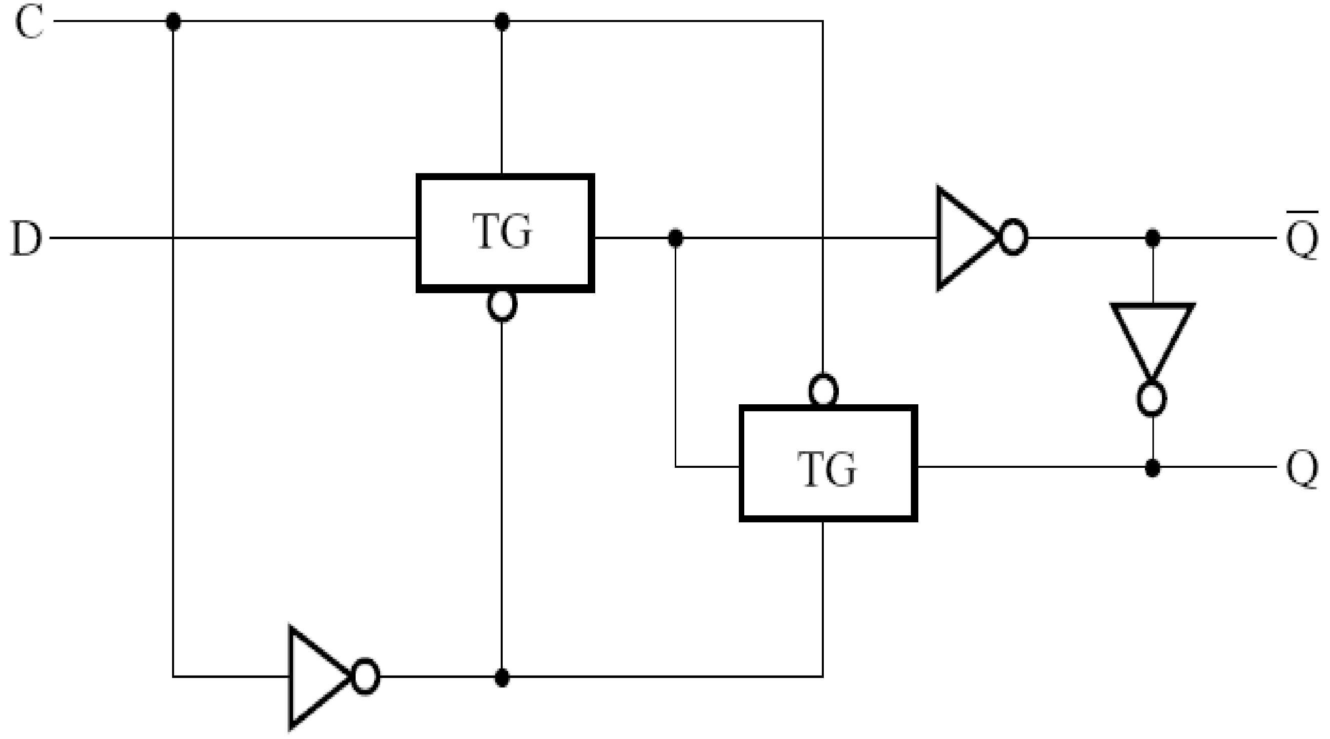


C	D	Q(t + 1)
0	X	No change
1	0	Q=0: Clear Q
1	1	Q=1: Set Q

The graphic symbol for a D Latch is:



D Latch with Transmission Gates

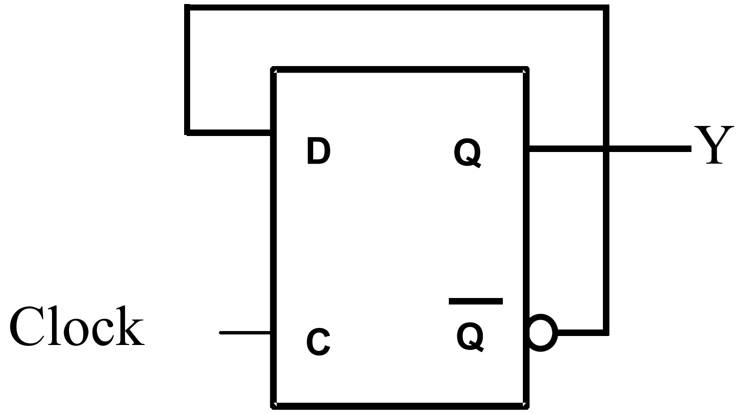


Flip-Flops

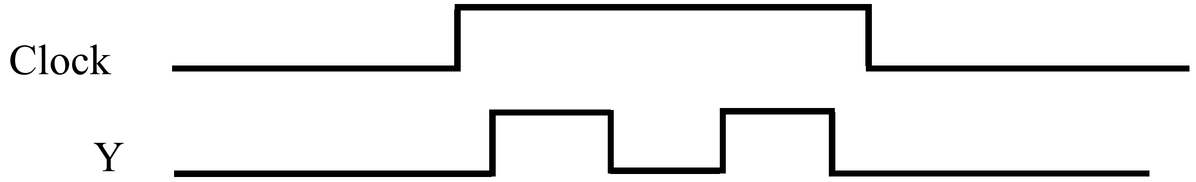
- The latch timing problem
- Master-slave flip-flop
- Edge-triggered flip-flop
- Standard symbols for storage elements
- Direct inputs to flip-flops
- Flip-flop timing

The Latch Timing Problem (continued)

- Consider the following circuit:



- Suppose that initially Y = 0.



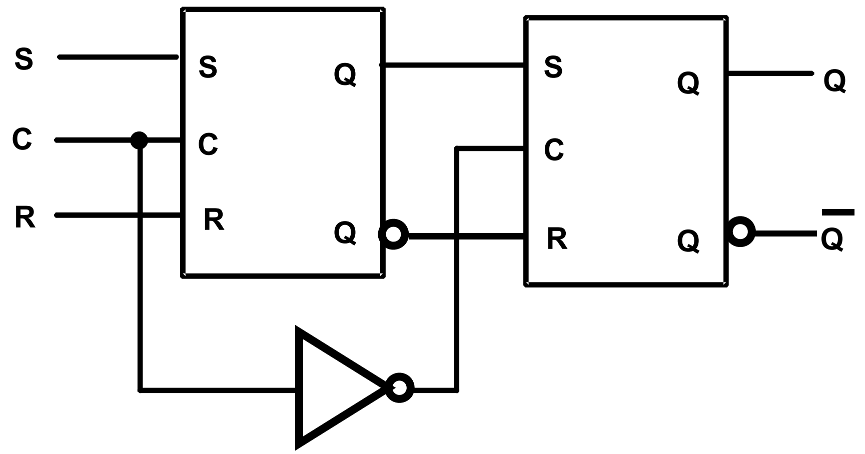
- As long as C = 1, the value of Y continues to change!
- The changes are based on the delay present on the loop through the connection from Y back to Y.
- This behavior is clearly unacceptable.
- <u>Desired behavior</u>: Y changes <u>only once</u> per clock pulse

The Latch Timing Problem (continued)

- A solution to the latch timing problem is to <u>break</u> the inner path from input to output within the storage element
- The commonly-used, path-breaking solutions are:
  - a master-slave flip-flop
  - an edge-triggered flip-flop

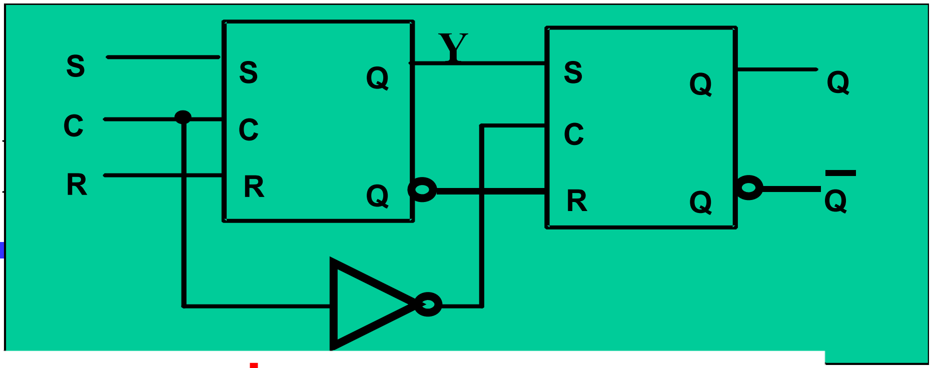
S-R Master-Slave Flip-Flop

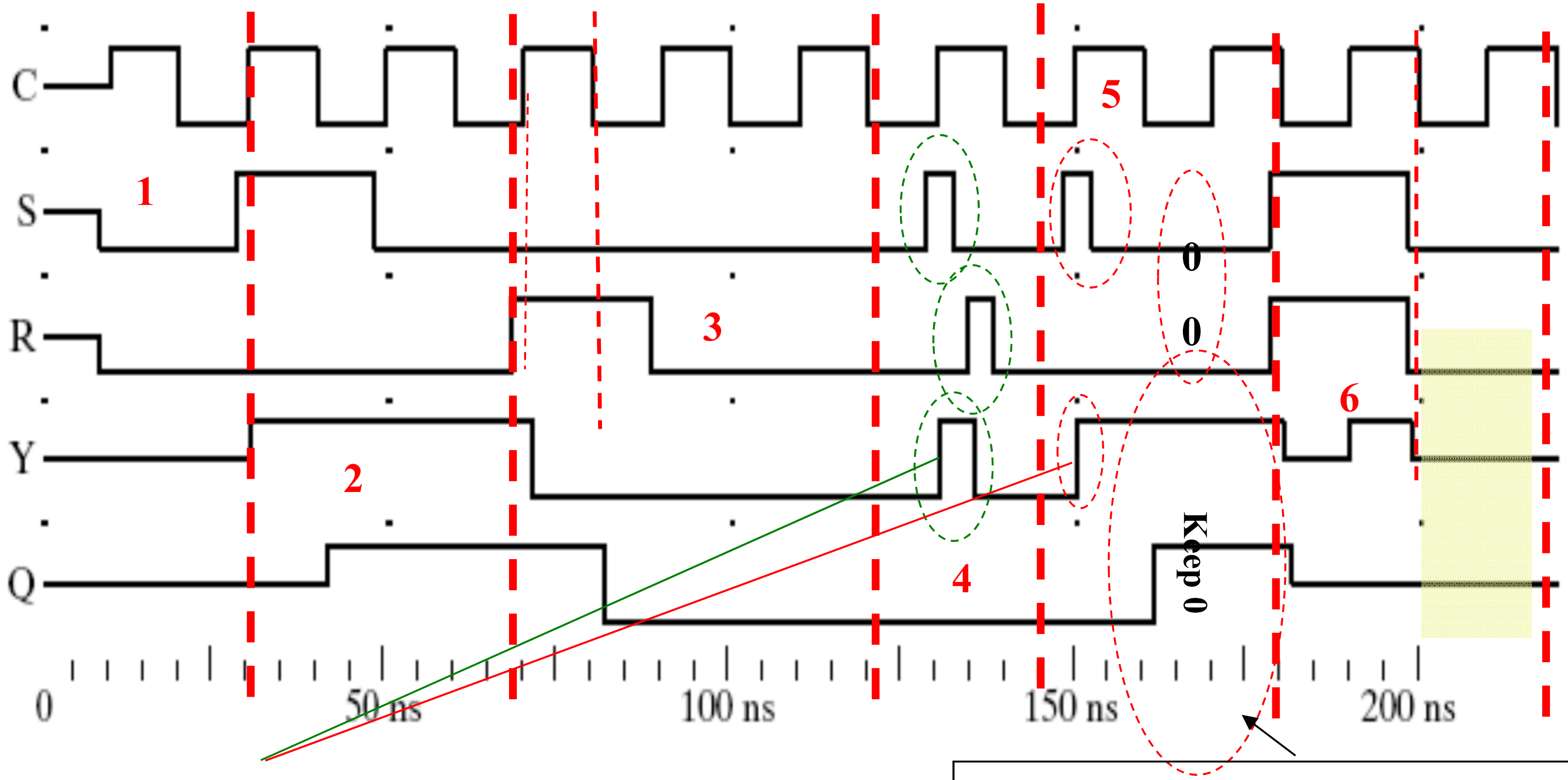
Consists of two clocked S-R latches in series with the clock on the second latch inverted



- The input is observed by the first latch when C = 1
- The output is changed by the second latch when C = 0
- The path from input to output is broken by the difference of the clocking values (C = 1 and C = 0).
- The behavior demonstrated in the previous example can be prevented since a change of Y based on D won't occur until the clock changes from 1 to 0.

S-R Master-Slave Flip-Flo





Disposable sampling

Reason: SR=11,
Input state is illegal

Before the arrival of pulse: Q=0
Before the end of the pulse: RS=00, Q should keep "0"

Flip-Flop Problem

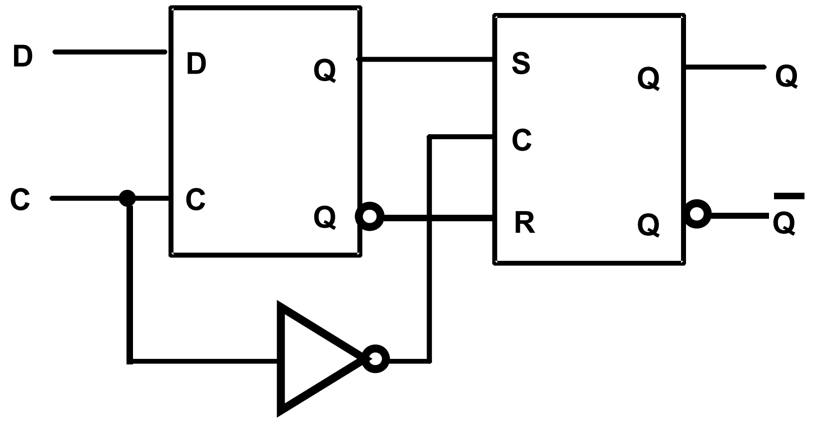
- The change in the flip-flop output is delayed by the pulse width ,which makes the circuit slower
- S and/or R are permitted to change while C = 1
  - Suppose Q = 0 and S goes to 1 and then back to 0 with R remaining at 0
    - The master latch sets to 1
    - A 1 is transferred to the slave
  - Suppose Q = 0 and S goes to 1 and back to 0 and R goes to 1 and back to 0
    - The master latch sets and then resets
    - A 0 is transferred to the slave
  - This behavior is called *1s catching*

Flip-Flop Solution

- Use edge-triggering instead of master-slave
- An *edge-triggered* flip-flop ignores the pulse while it is at a constant level and triggers only during a transition of the clock signal
- Edge-triggered flip-flops can be built directly at the electronic circuit level, or
- A master-slave D flip-flop which also exhibits edge-triggered behavior can be used.

Edge-Triggered D Flip-Flop

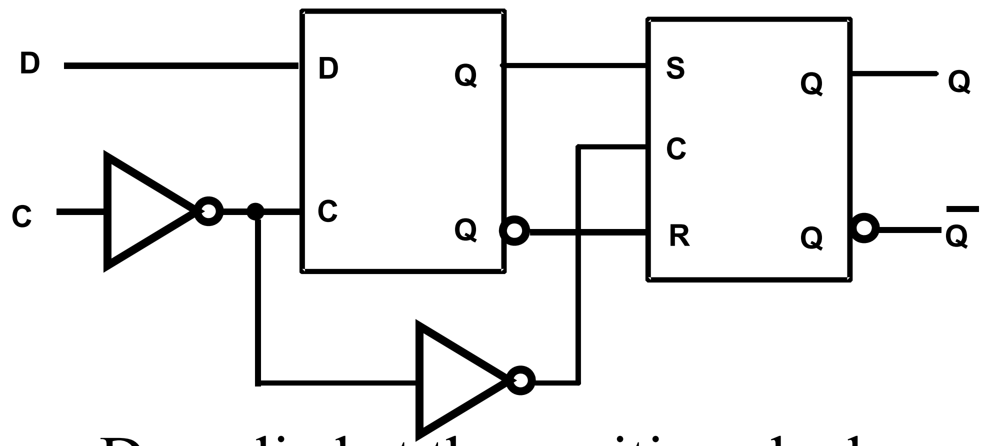
The function of edge-triggered D flip-flop is the same as the master-slave D flip-flop



- It can be formed by:
  - Replacing the first clocked S-R latch with a clocked D latch or
  - Adding a D input and inverter to a master-slave S-R flip-flop
- The delay of the S-R master-slave flip-flop can be avoided since the 1s-catching behavior is not present with D replacing S and R inputs
- The change of the D flip-flop output is associated with the negative edge at the end of the pulse
- It is called a *negative-edge triggered* flip-flop

Positive-Edge Triggered D Flip-Flop

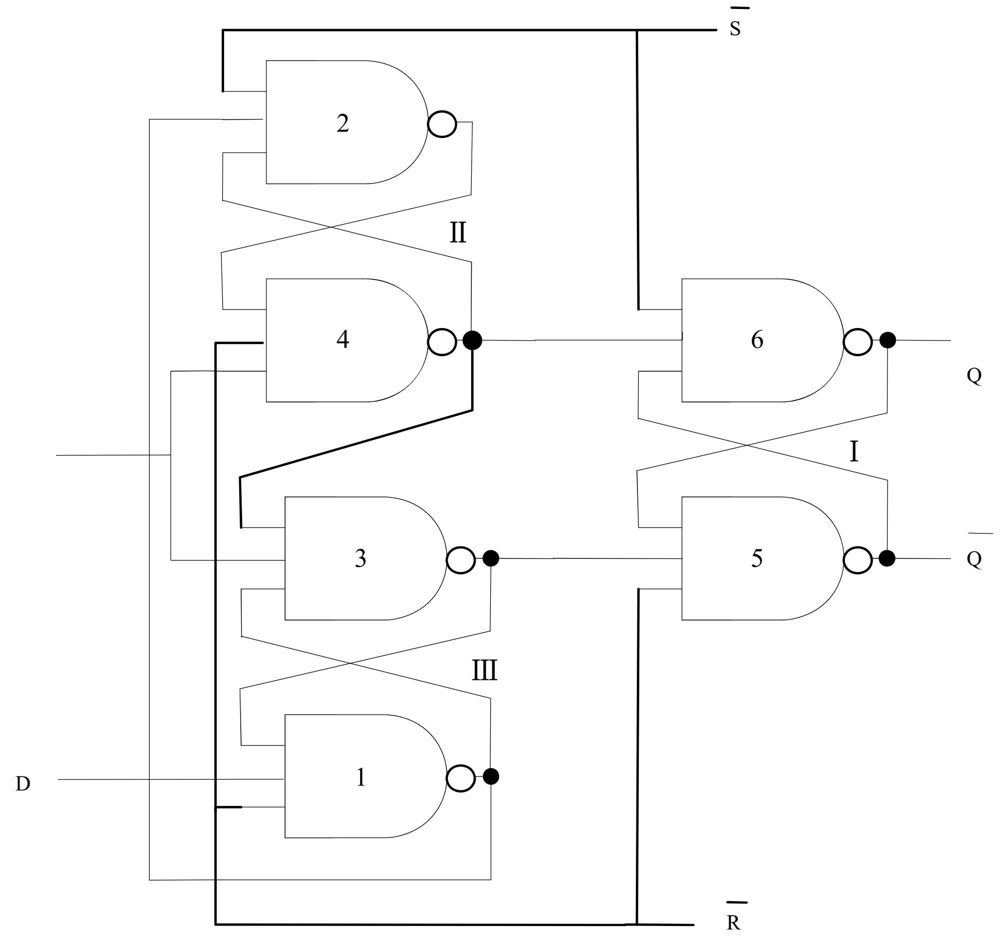
- Formed by adding inverter to clock input



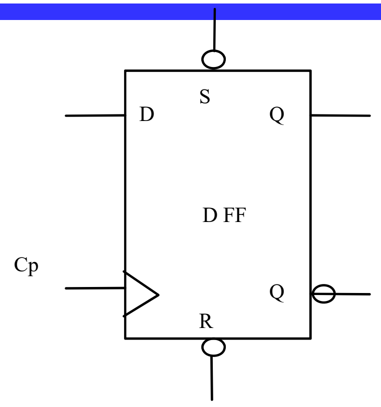
- Q changes to the value on D applied at the positive clock edge within timing constraints to be specified
- Our choice as the <u>standard flip-flop</u> for most sequential circuits
- Q(t) is the current output of flip-flop while D(t) is the current input, and Q(t+1) represent the next output, then we have:
- Q(t+1) = D(t)

Positive-Edge-Triggered D Flip-Flop

Circuit for Problem 5-3



(a) Logic Circuit



(b) Logic Symbol

Asynchronous		**Positive-Edge-Triggered**			
$\overline{R}$	$\overline{S}$	Cp	D	Q	$\overline{Q}$
0	1	X	X	0	1
1	0	X	X	1	0
1	1	↑	0	0	1
1	1	↑	1	1	0

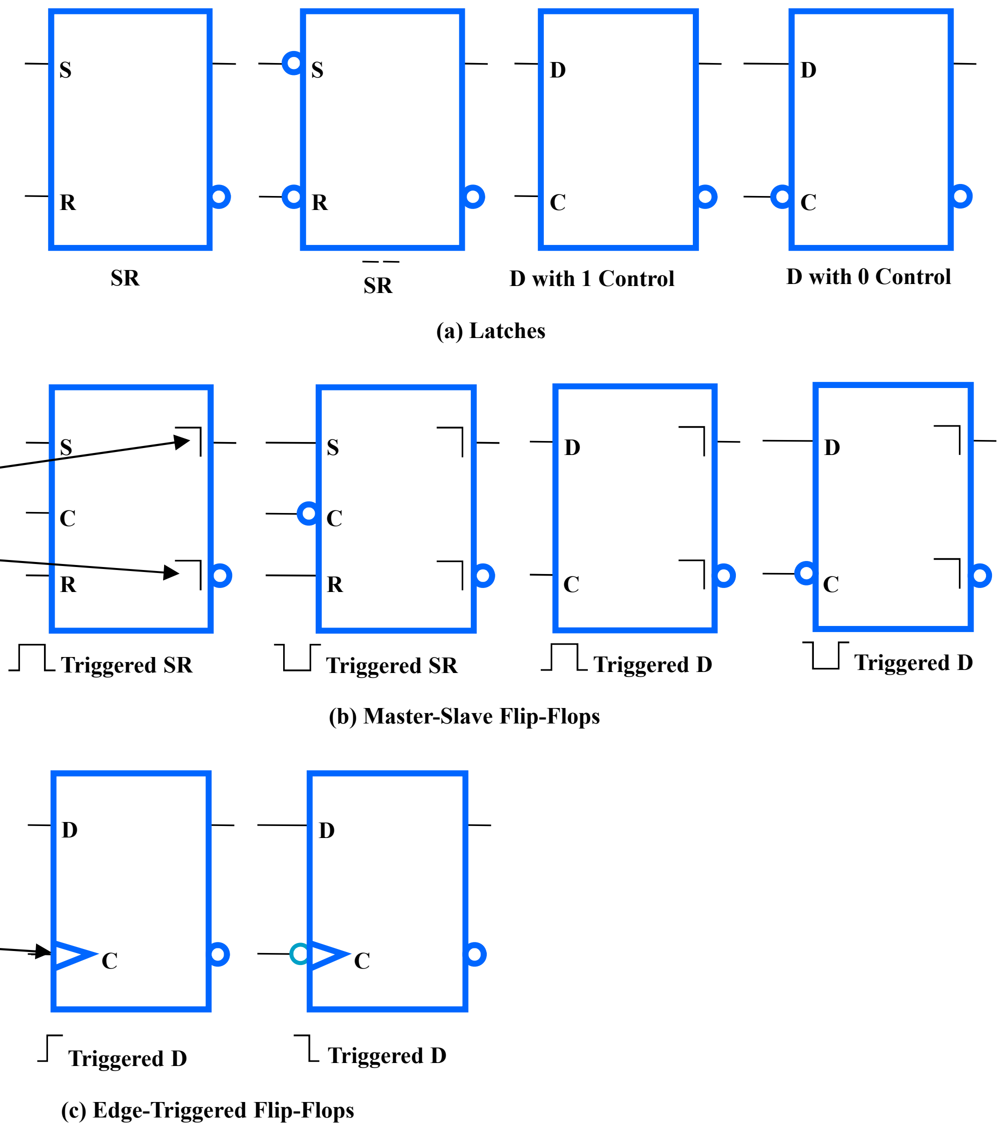
(c) Function Table

Standard Symbols for Storage Elements

- Master-Slave:

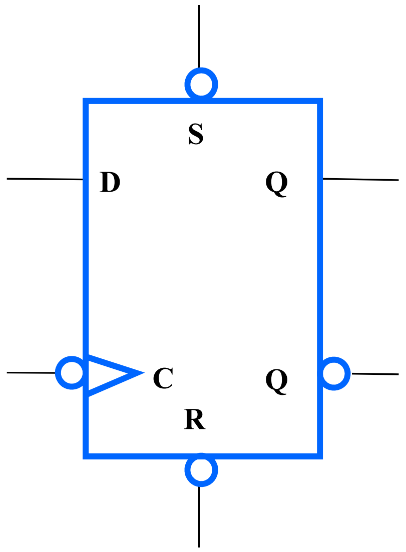
Postponed output indicators

- Edge-Triggered: Dynamic indicator



Direct Inputs

- At power up or at reset, all or part of a sequential circuit usually is initialized to a known state before it begins operation
- This initialization is often done outside of the clocked behavior of the circuit, i.e., asynchronously.



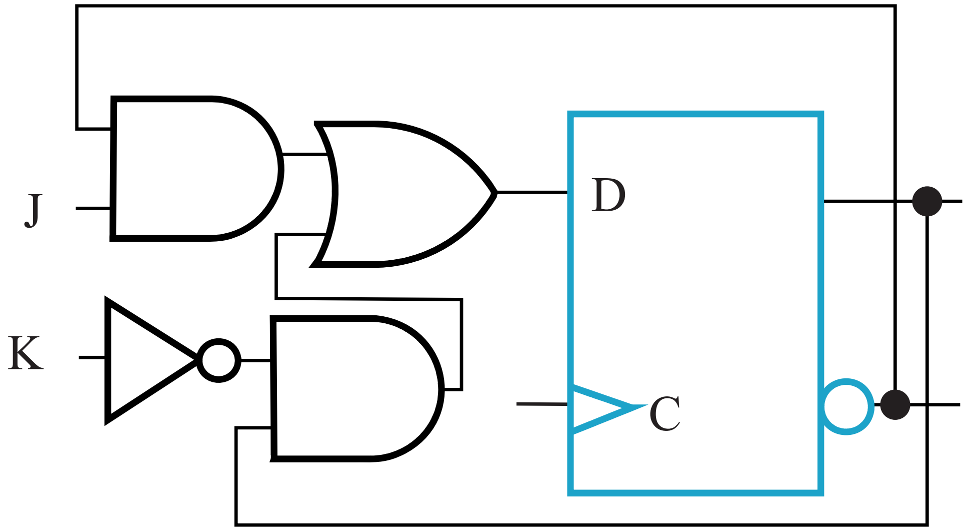
- Direct R and/or S inputs that control the state of the latches within the flip-flops are used for this initialization.
- For the example flip-flop shown
  - When R is 0, resets the flip-flop to the 0 state
  - When S is 0, sets the flip-flop to the 1 state
  - When R and S are both 1, flip-flop works normally
  - State undefined when R and S are both set to 0

J-K Flip-flop

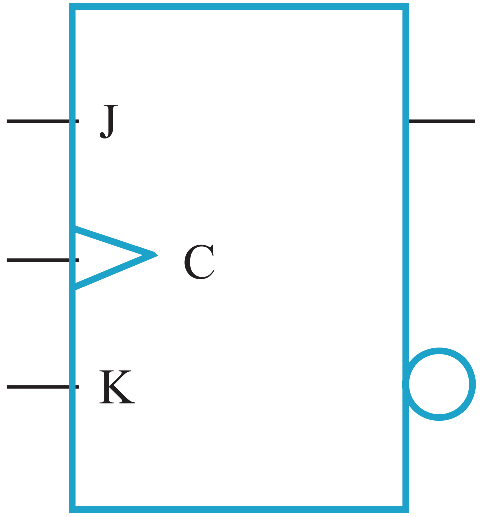
- Behavior
  - Same as S-R flip-flop with J analogous to S and K analogous to R
  - Except that J = K = 1 is allowed, and
  - For J = K = 1, the flip-flop changes to the *opposite state*
  - As a master-slave, has same “1s catching” behavior as S-R flip-flop
  - If the master changes to the wrong state, that state will be passed to the slave
    - E.g., if master falsely set by J = 1, K = 1 cannot reset it during the current clock cycle

J-K Flip-flop (continued)

- Implementation
  - To avoid 1s catching behavior, one solution used is to use an edge-triggered D as the core of the flip-flop



- 符号

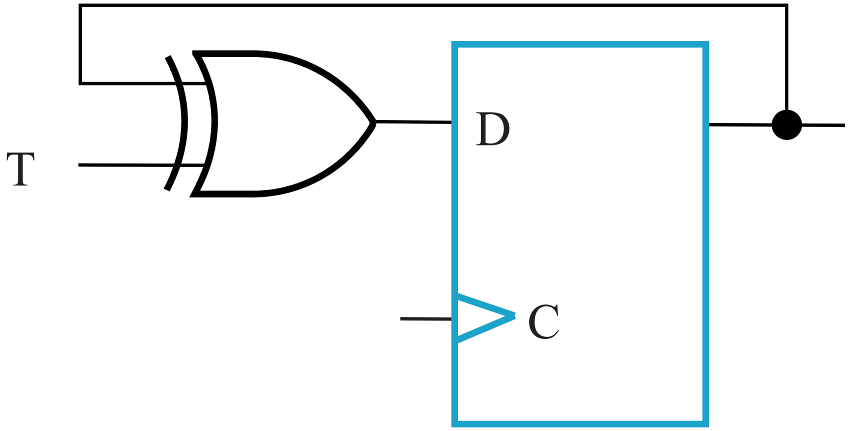


T Flip-flop

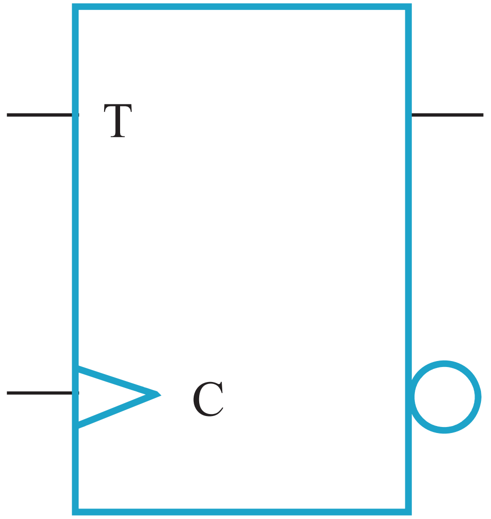
- Behavior
  - Has a single input T
    - For T = 0, no change to state
    - For T = 1, changes to opposite state
- Same as a J-K flip-flop with J = K = T
- As a master-slave, has same “1s catching” behavior as J-K flip-flop
- Cannot be initialized to a known state using the T input
  - Reset (asynchronous or synchronous) essential

T Flip-flop (continued)

- Implementation
  - To avoid 1s catching behavior, one solution used is to use an edge-triggered D as the core of the flip-flop

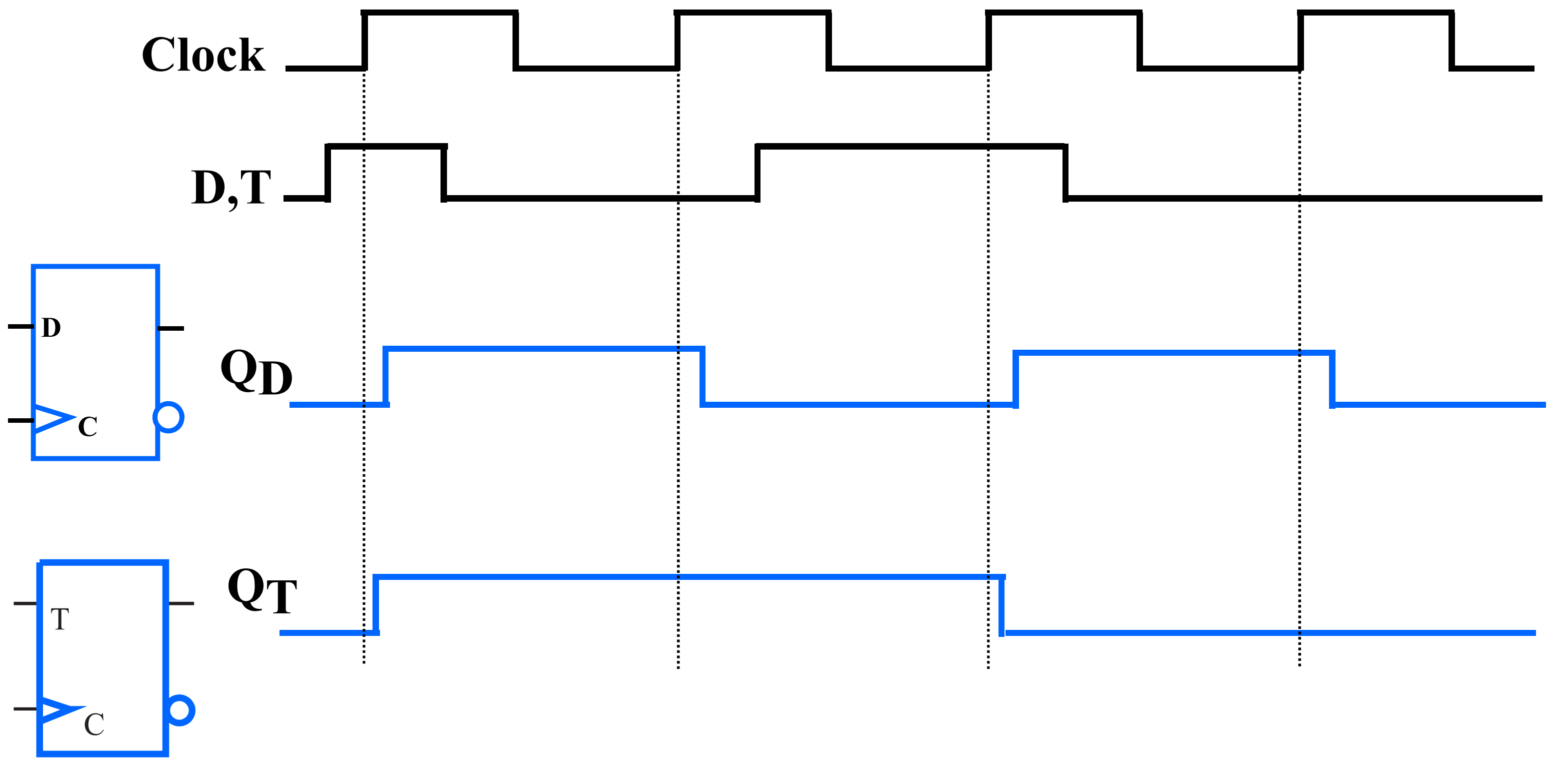


- 符号



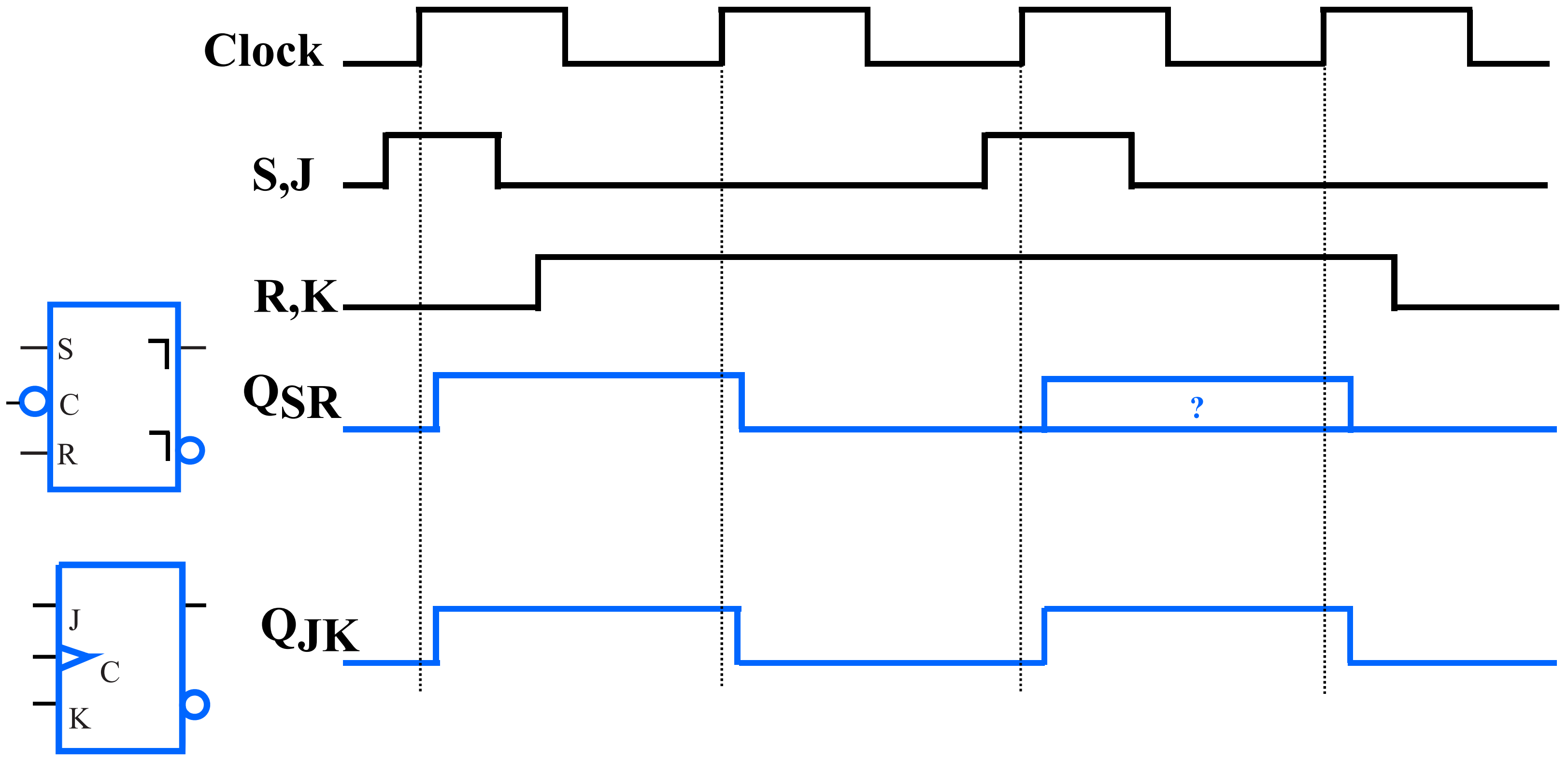
Flip-flop Behavior Example

- Find the output waveforms for the flip-flops shown:



Flip-Flop Behavior Example (continued)

- Find the output waveforms for the flip-flops shown:

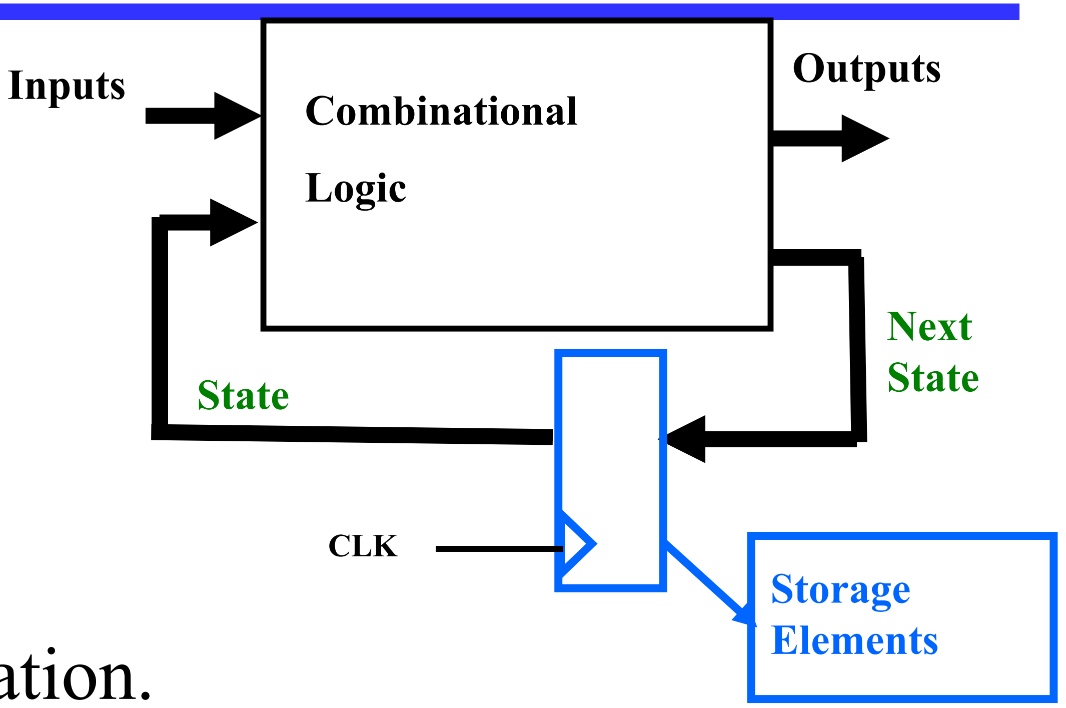


Overview

- Introduction of sequential circuits
- Basic storage elements
- Synchronous sequential logic analysis
- Synchronous sequential logic design
- Classic sequential logic circuits

Sequential Circuit Analysis

- General Model



  - Flip-flop input equation.
  - Next State equation: next state at time (t+1) is a Boolean function of State and Inputs.
  - Output function: a Boolean function of State (t) and (sometimes) Inputs (t).

Sequential Circuit Analysis

- Sequential Circuit Analysis is an procedure of specifying the logic diagram of a given sequential circuit. A state table and state diagram are presented to describe the behavior of the circuit, demonstrate the time sequence of inputs, outputs and states, and illustrate the functionality of the given circuits.

Sequential Circuit Analysis Procedure

1. Derive the output equations，flip-flop input equations and next state functions
2. Derive the truth table with state:

   Inputs: inputs of circuit, present state of the circuit

   Outputs: outputs of circuit, next state of all flip-fops
3. List the states of the sequential circuit
4. Obtain a state diagram
5. Analyze the external performance of the circuit
6. Verify the correctness of the circuit, check the self-recovery capability and draw the timing parameters

Example 1

- **Input:** x(t)
- **Output:** y(t)
- **State:** (A(t), B(t))
- **What is the Output Function?**

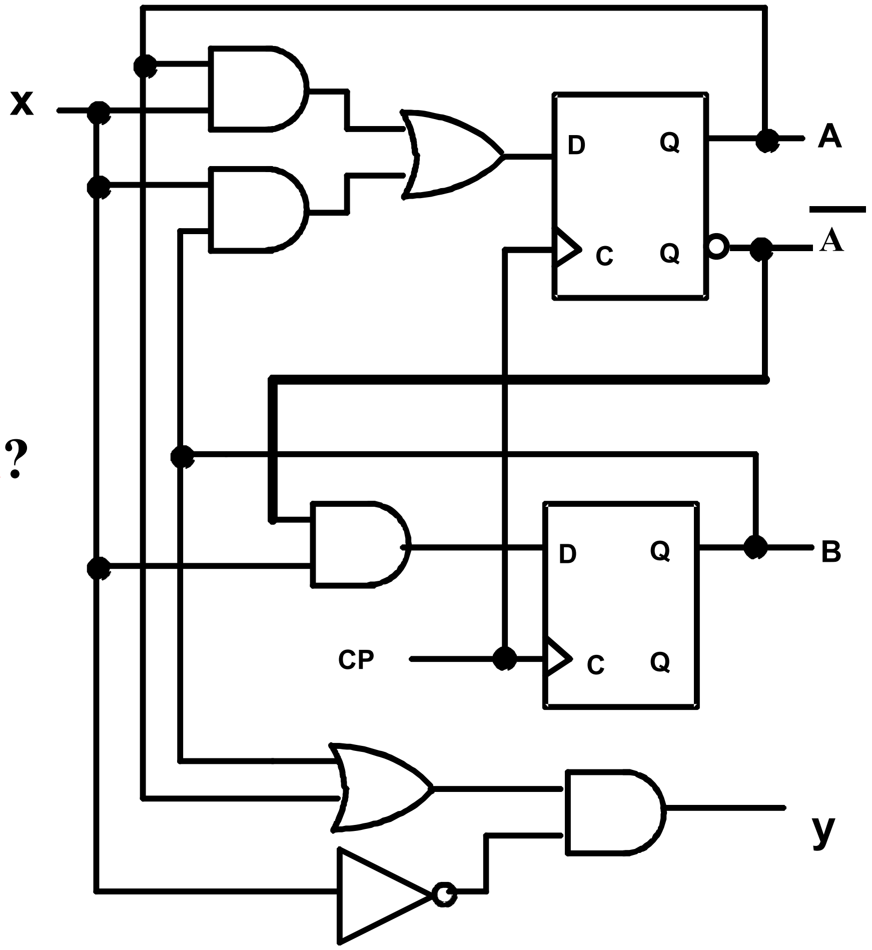
  y=
- **What is the Trigger input Function?**

  D_A=

  D_B=
- **What is the Next State Function?**

  A(t+1)=

  B(t+1)=



Example 1 (continued)

- Flip-flop Input functions:

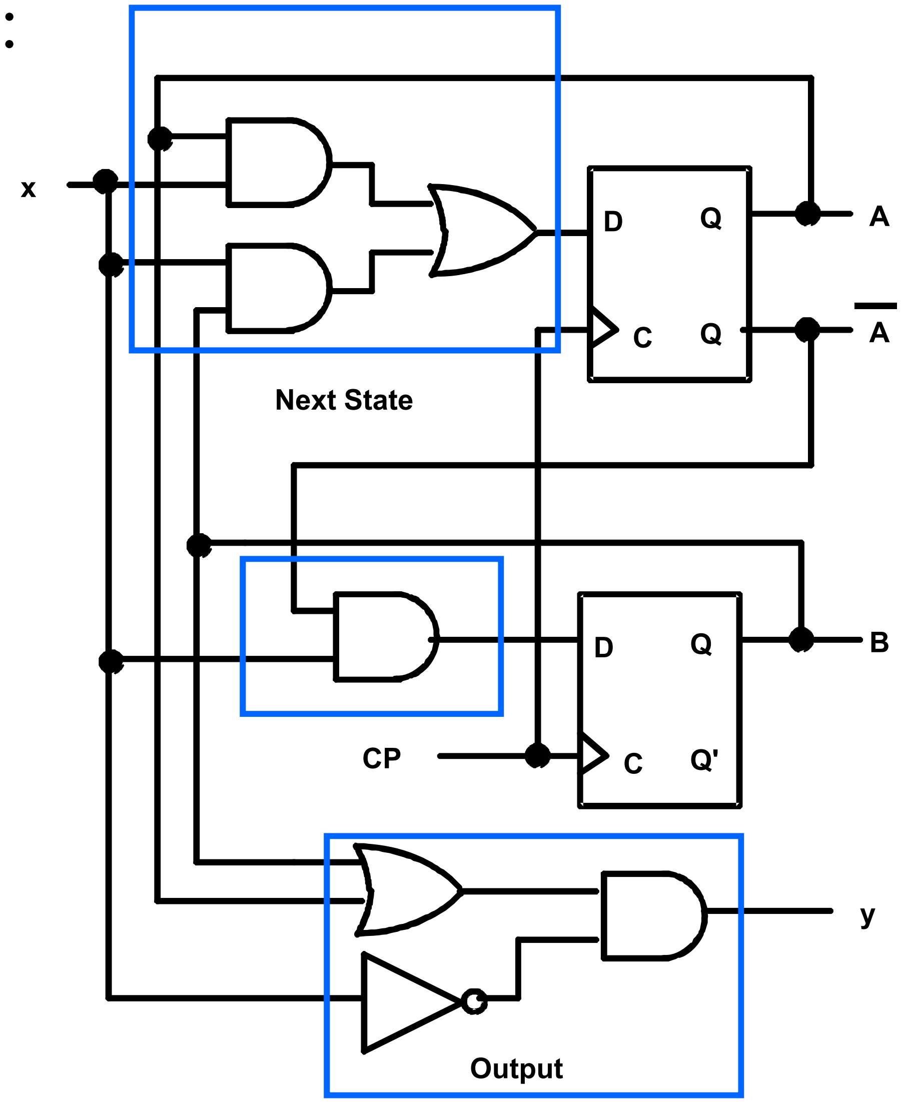
  $D_A = A(t)x(t) + B(t)x(t)$

  $D_B = \overline{A}(t)x(t)$

- Next State equations:
  - $A(t+1) = D_A$
  - $B(t+1) = D_B$

- Output:

  $y(t) = \overline{x}(t)(B(t) + A(t))$



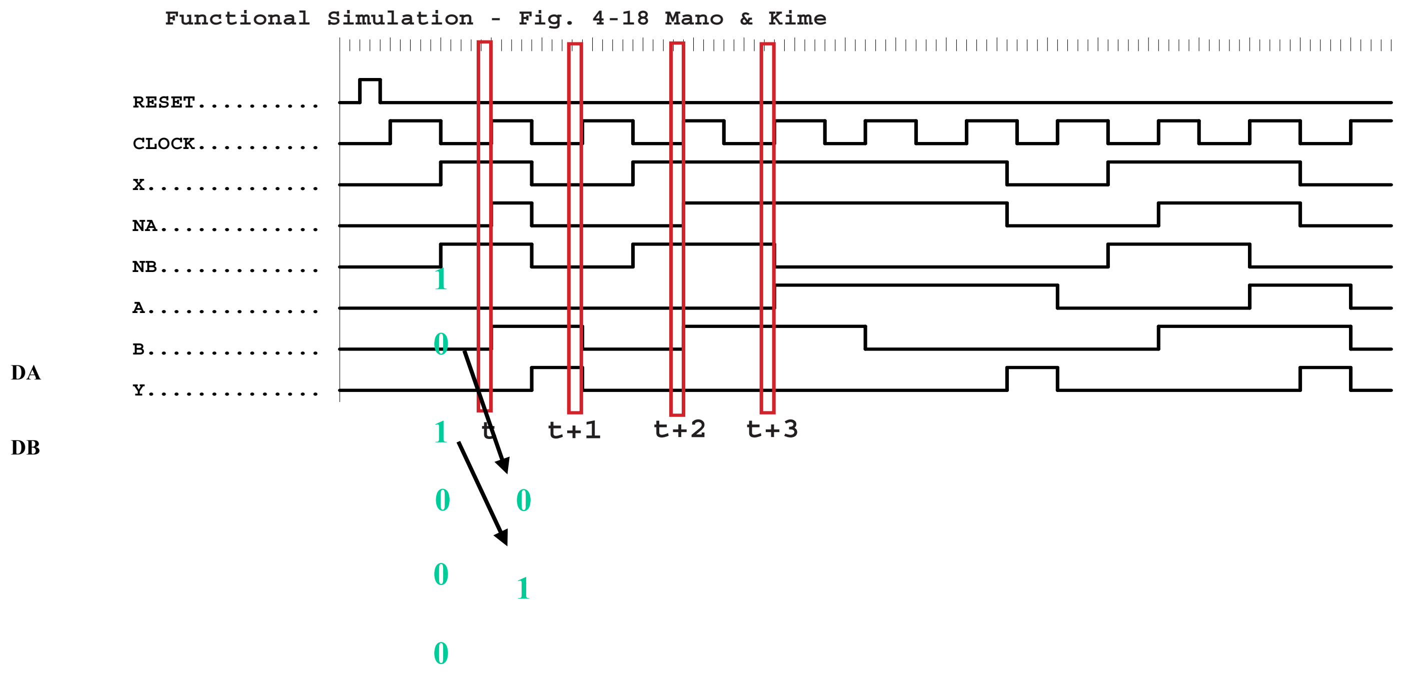
Example 1 (continued)

$DA = A(t)x(t) + B(t)x(t)$

$DB = \overline{A}(t)x(t)$

$y(t) = \overline{x}(t)(B(t) + A(t))$

- Where in time are inputs, outputs and states defined?



State Table Characteristics

- *State table* – a multiple variable table with the following four sections:
  - *Present State* – the values of the state variables for each allowed state.
  - *Input* – the input combinations allowed.
  - *Next-state* – the value of the state at time (t+1) based on the present state and the input.
  - *Output* – the value of the output as a function of the present state and (sometimes) the input.
- From the viewpoint of a truth table:
  - the inputs are Input, Present State
  - and the outputs are Output, Next State

Example 1: State Table (from Fig. 5-15)

- The state table can be filled in using the next state and output equations:
  - $A(t+1) = A(t)x(t) + B(t)x(t)$
    $B(t+1) = \overline{A}(t)x(t)$
  - $y(t) = \overline{x}(t)(B(t) + A(t))$

Present State		Input	Next State		Output
A(t)	B(t)	x(t)	A(t+1)	B(t+1)	y(t)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Example 1: Alternative State Table

- 2-dimensional table that matches well to a K-map. Present state rows and input columns in Gray code order.
  - $A(t+1) = A(t)x(t) + B(t)x(t)$
  - $B(t+1) = \overline{A}(t)x(t)$
  - $y(t) = \overline{x}(t)(B(t) + A(t))$

Present State	**Next State**		**Output**	
	x(t)=0	**x(t)=1**	**x(t)=0**	**x(t)=1**
A(t) B(t)	**A(t+1)B(t+1)**	**A(t+1)B(t+1)**	**y(t)**	**y(t)**
0 0	**0 0**	**0 1**	**0**	**0**
0 1	**0 0**	**1 1**	**1**	**0**
1 1	**0 0**	**1 0**	**1**	**0**
1 0	**0 0**	**1 0**	**1**	**0**

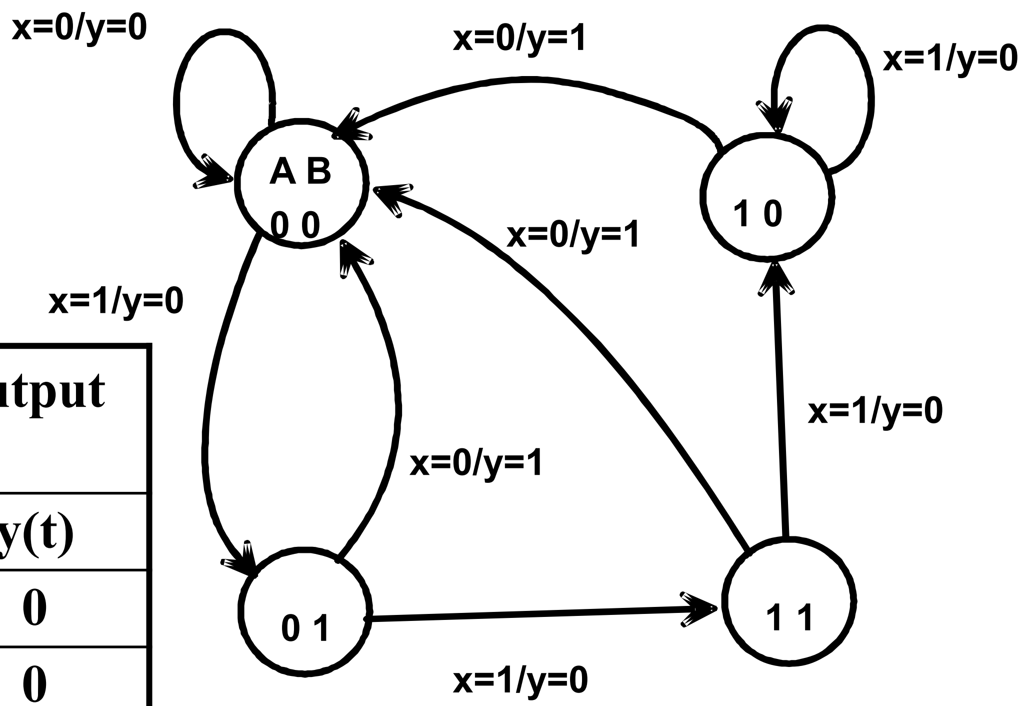
State Diagrams

- The sequential circuit function can be represented in graphical form as a state diagram with the following components:
  - A circle with the state name in it for each state
  - A directed arc from the Present State to the Next State for each state transition
  - A label on each directed arc with the Input values which causes the state transition, and
  - A label:
    - On each circle with the output value produced, or
    - On each directed arc with the output value produced.

Example 1: State Diagram

- Which type?

Present State		Input	Next State		Output
A(t)	B(t)	x(t)	A(t+1)	B(t+1)	y(t)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0



Moore and Mealy Models

- There are two formal models of FSMs:

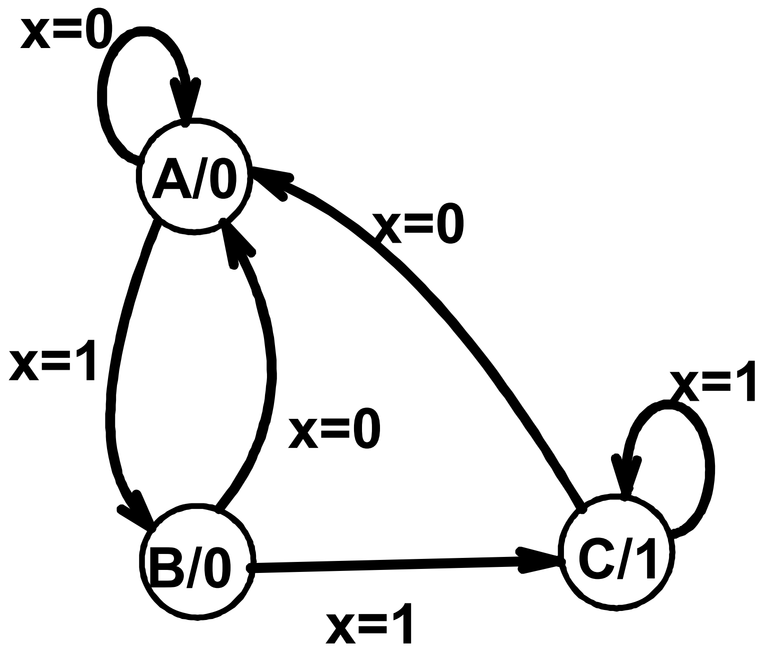
- Moore model
  - Named after E.F. Moore
  - Outputs are a function ONLY of states
  - Usually specified on the states.

- Mealy model
  - Named after G. Mealy
  - Outputs are a function of inputs AND states
  - Usually specified on the state transition arcs.

Moore and Mealy Example

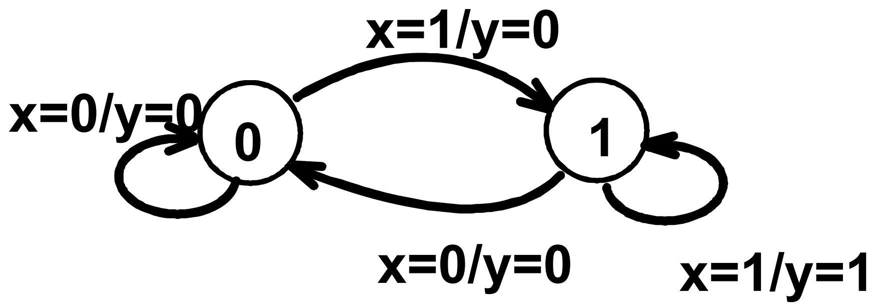
- Moore model state table/diagram maps **states to outputs**

Present State	Next State x=0	Next State x=1	Output
A	A	B	0
B	A	C	0
C	A	C	1



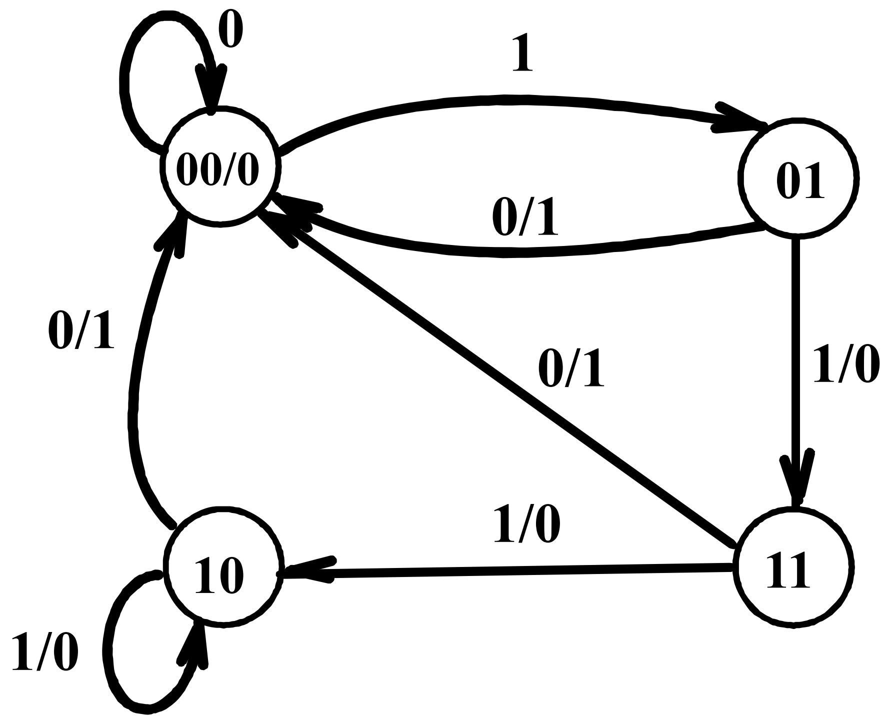
- Mealy model state table/diagram maps **inputs and state to outputs**

Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
0	0	1	0	0
1	0	1	0	1



Mixed Moore and Mealy Outputs

- In real designs, some outputs may be Moore type and other outputs may be Mealy type.
- Example: This figure can be used to illustrate this mixed model
  - State 00: Moore
  - States 01, 10, and 11: Mealy
- Simplifies output specification

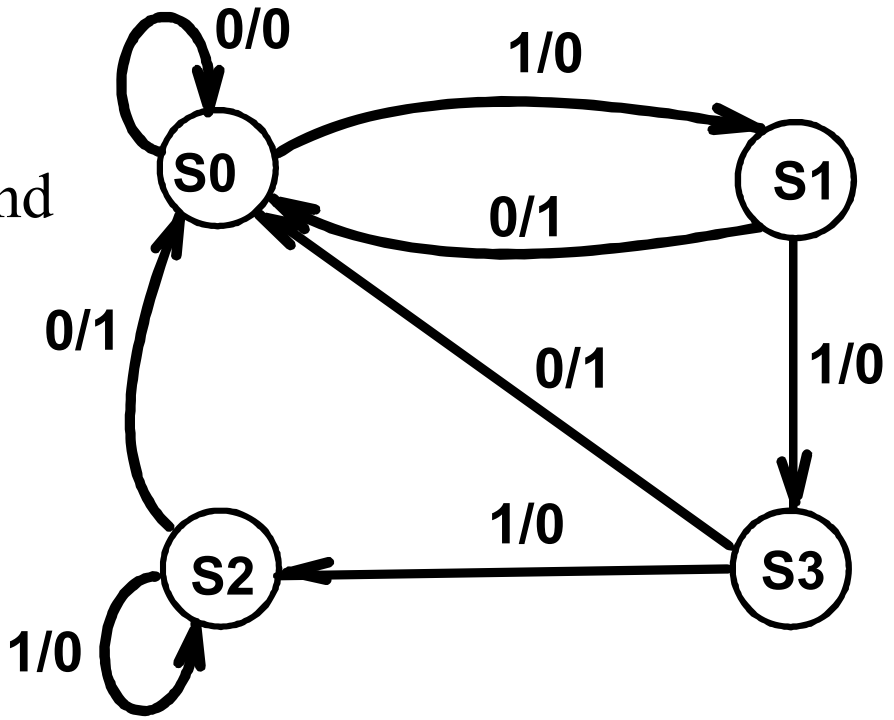


Equivalent State

- Two states are ***equivalent*** if their response for each possible input sequence is an identical output sequence.
- Alternatively, two states are ***equivalent*** if their outputs produced for each input symbol is identical and their next states for each input symbol are the same or equivalent.

Equivalent State Example

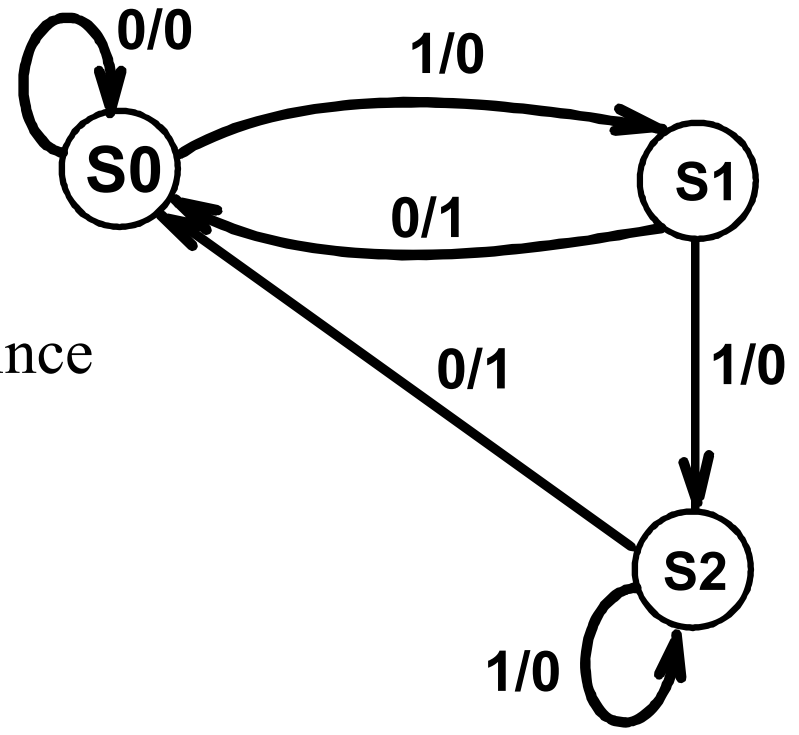
- For states S3 and S2
  - the output for input 0 is 1 and input 1 is 0, and
  - the next state for input 0 is S0 and for input 1 is S2

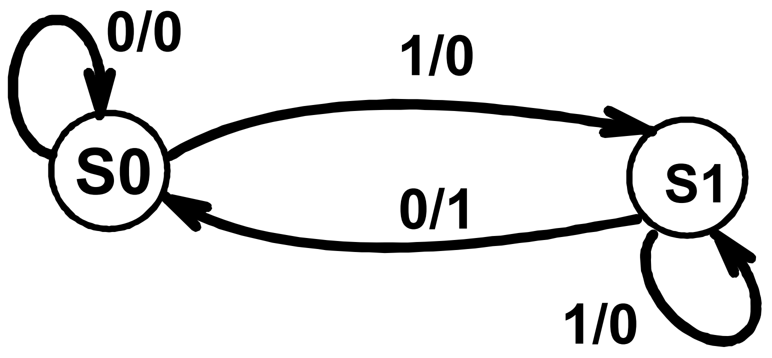


- By the alternative definition, states S3 and S2 are equivalent.

Equivalent State Example

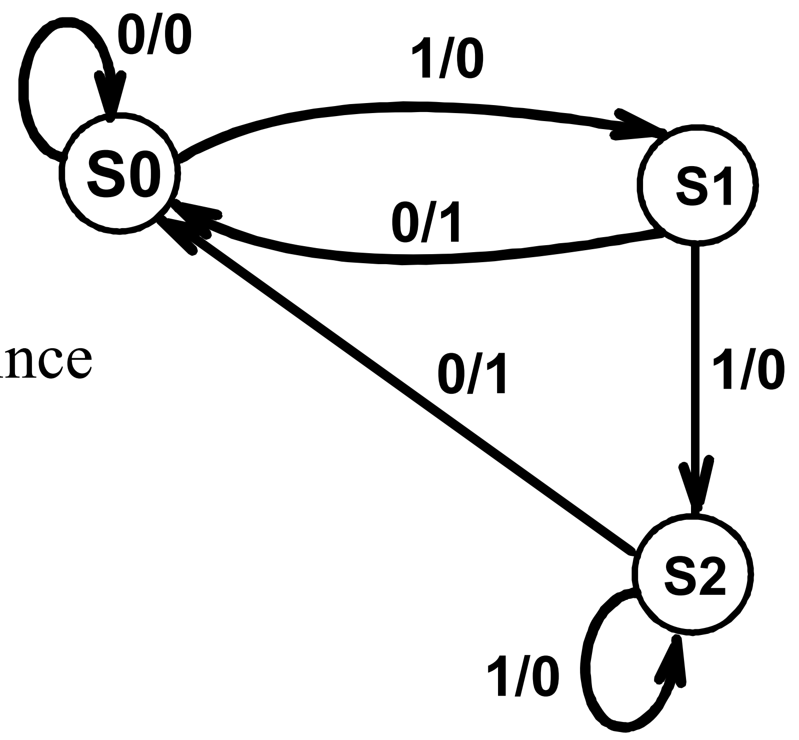
- Replacing S3 and S2 by a single state gives state diagram:
- Examining the new diagram, states S1 and S2 are equivalent since
  - their outputs for input 0 is 1 and input 1 is 0
  - their next state for input 0 is S0 and for input 1 is S2
- Replacing S1 and S2 by a single state gives state diagram:

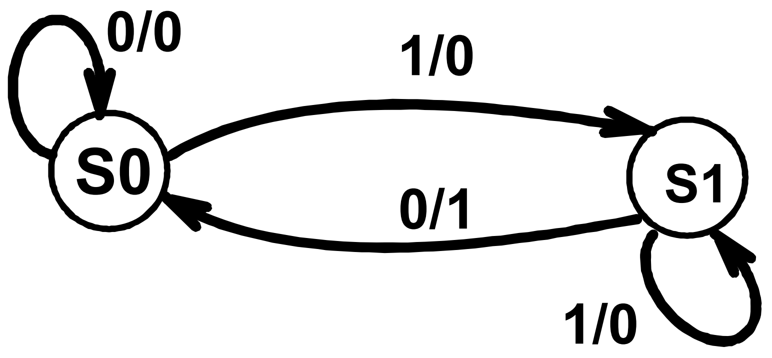




Equivalent State Example

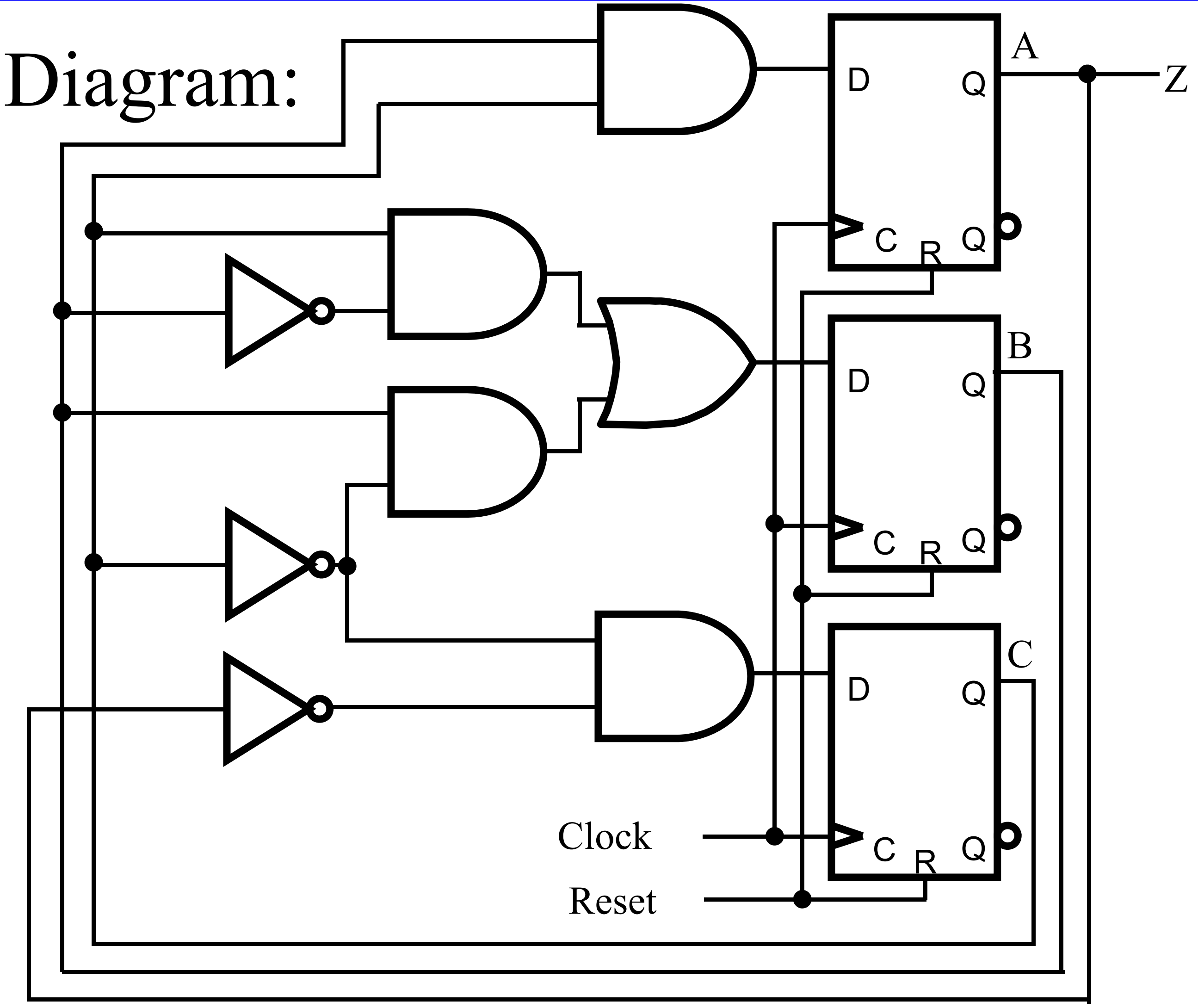
- Replacing S3 and S2 by a single state gives state diagram:
- Examining the new diagram, states S1 and S2 are equivalent since
  - their outputs for input 0 is 1 and input 1 is 0
  - their next state for input 0 is S0 and for input 1 is S2
- Replacing S1 and S2 by a single state gives state diagram:





Example 2: Sequential Circuit Analysis

- Logic Diagram:



Example 2: Flip-Flop Input Equations

- Variables
  - Inputs: None
  - Outputs: Z
  - State Variables: A, B, C
- Initialization: Reset to (0,0,0)
- Equations
  - $A(t+1) =$ $\qquad\qquad$ $Z =$
  - $B(t+1) =$
  - $C(t+1) =$

Example 2: Flip-Flop Input Equations

- Variables
  - Inputs: None
  - Outputs: Z
  - State Variables: A, B, C
- Initialization: Reset to (0,0,0)
- Equations
  - $A(t+1) = B(t)C(t)$ $\qquad Z = A(t)$
  - $B(t+1) = \overline{B}(t)C(t) + B(t)\overline{C}(t)$
  - $C(t+1) = \overline{A}(t)\overline{C}(t)$

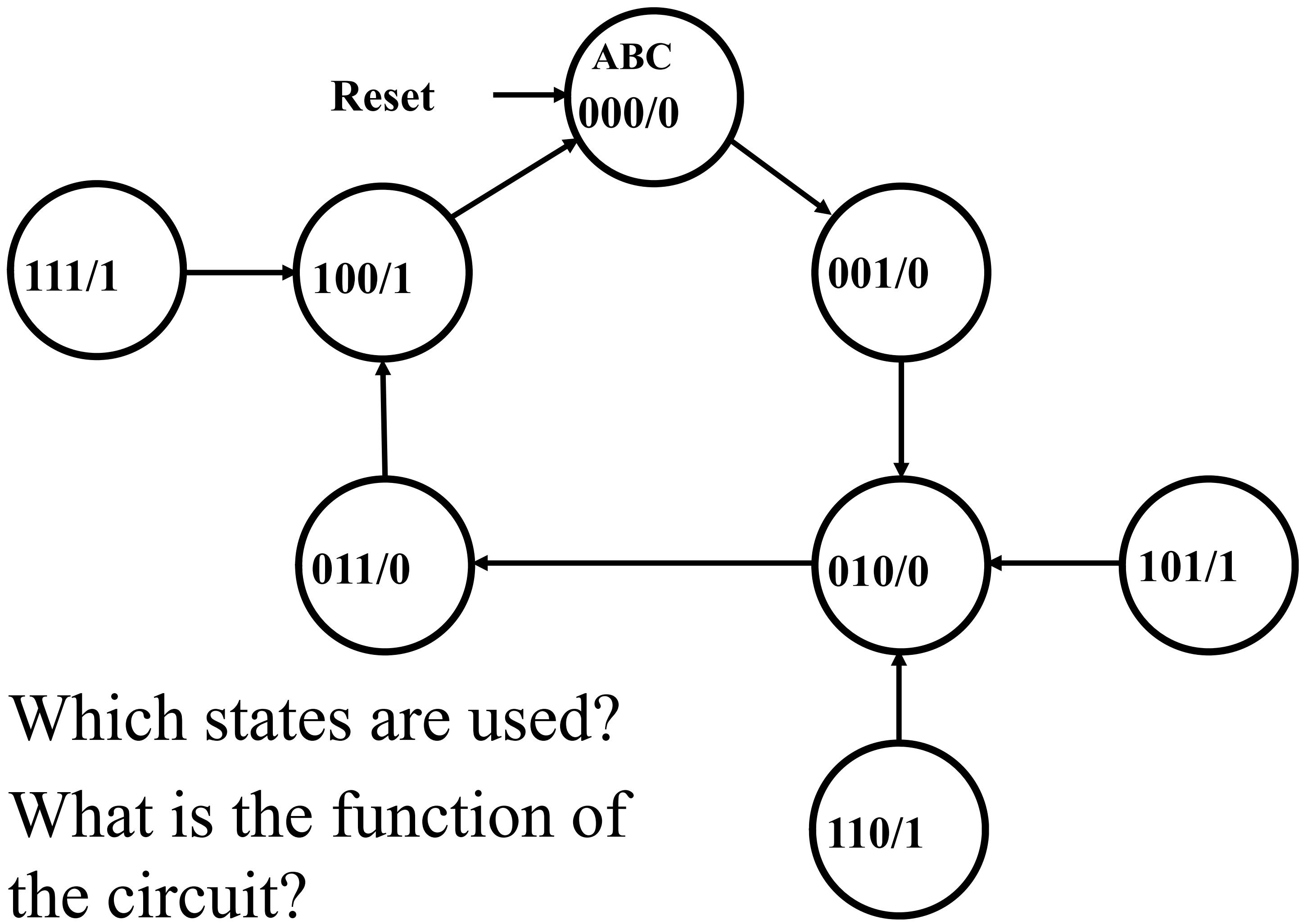
Example 2: State Table

A B C	A'B'C'	Z
0 0 0		
0 0 1		
0 1 0		
0 1 1		
1 0 0		
1 0 1		
1 1 0		
1 1 1		

Example 2: State Table

A B C	A’B’C’	Z
0 0 0	0 0 1	0
0 0 1	0 1 0	0
0 1 0	0 1 1	0
0 1 1	1 0 0	0
1 0 0	0 0 0	1
1 0 1	0 1 0	1
1 1 0	0 1 0	1
1 1 1	1 0 0	1

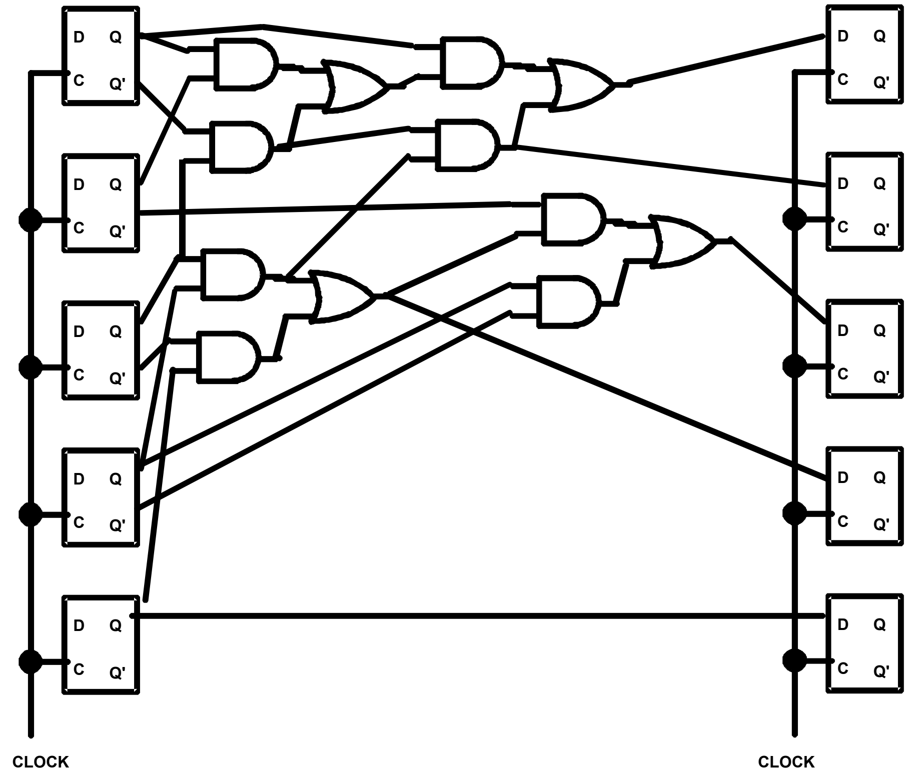
Example 2: State Diagram



- Which states are used?
- What is the function of the circuit?
- Does it have self-recovery capability?

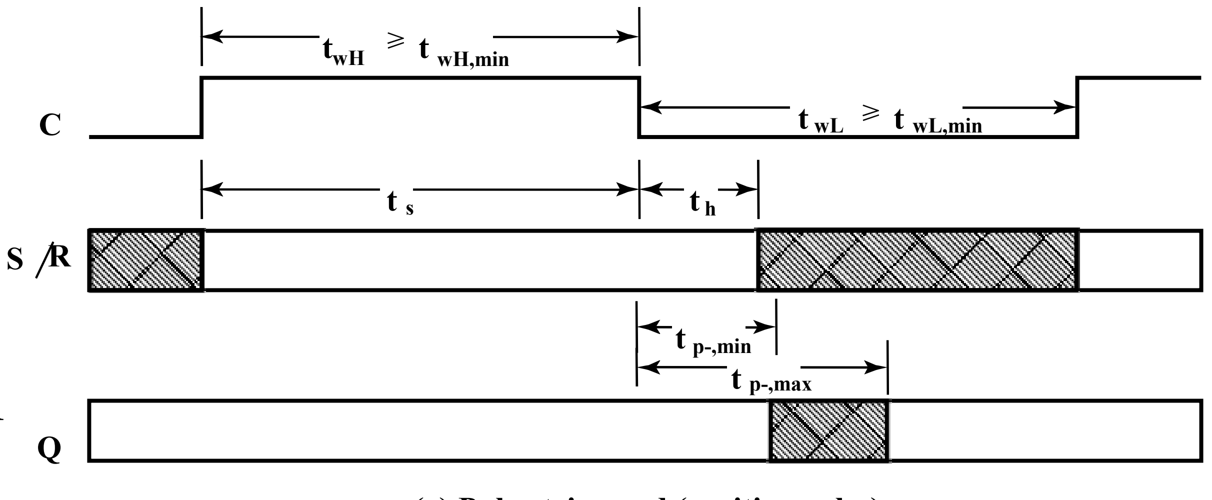
Circuit and System Level Timing

- Consider a system consists of flip-flops connected by logic:
- If the clock period is too short, data changes may not be able to propagate through the circuit to flip-flop inputs before the setup time interval begins

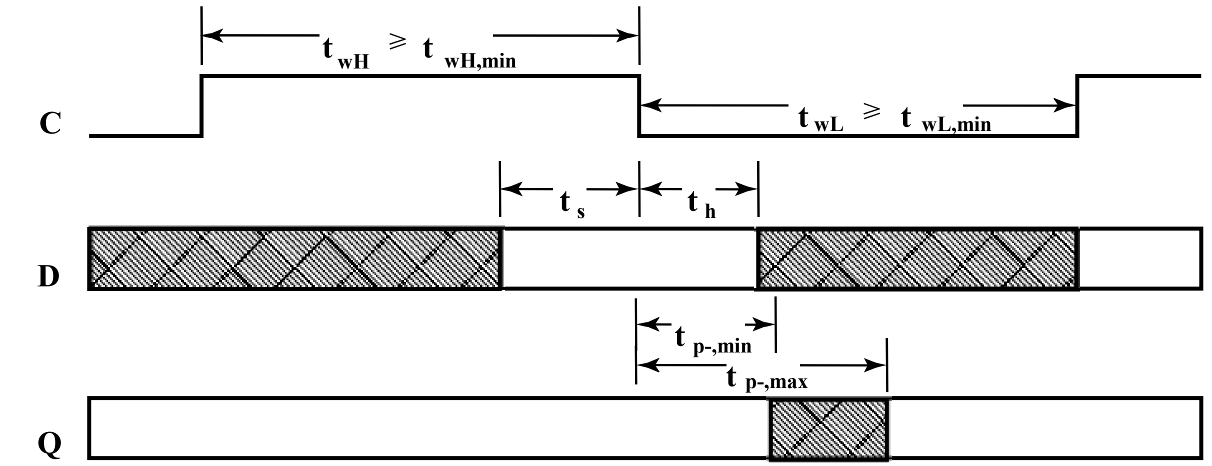


Flip-Flop Timing Parameters

- t_s - setup time
- t_h - hold time
- t_w – clock pulse width
- t_{px} - propagation delay
  - t_{PHL} - High-to-Low
  - t_{PLH} - Low-to-High
  - t_{pd} - max (t_{PHL}, t_{PLH})



(a) Pulse-triggered (positive pulse)



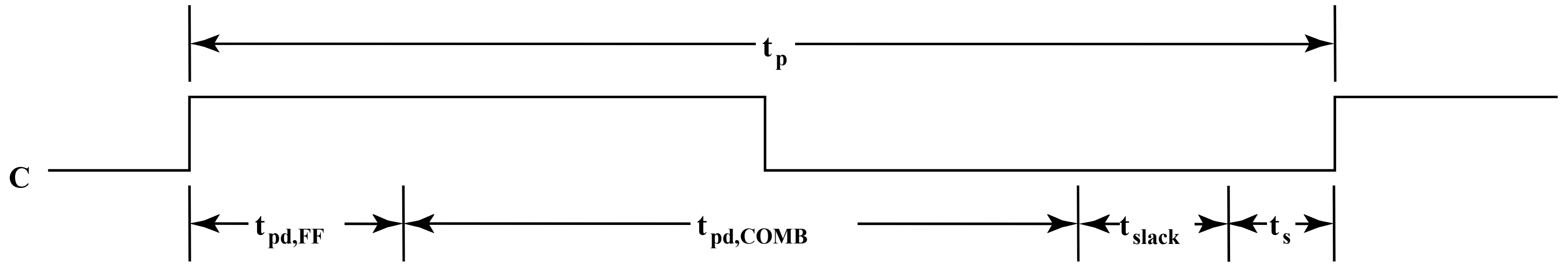
(b) Edge-triggered (negative edge)

Flip-Flop Timing Parameters (continued)

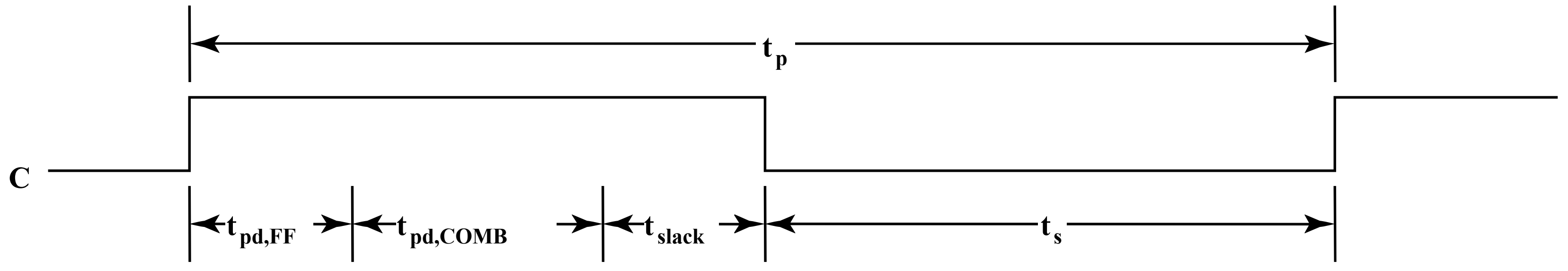
- t_s - setup time
  - Master-slave - Equal to the width of the triggering pulse
  - Edge-triggered - Equal to a time interval that is generally much less than the width of the the triggering pulse
- t_h - hold time - Often equal to zero
- t_{px} - propagation delay
  - Same parameters as for gates except
  - Measured from clock edge that triggers the output change to the output change

Circuit and System Level Timing (continued)

- Timing components along a path from flip-flop to flip-flop



(a) Edge-triggered (positive edge)



(b) Pulse-triggered (negative pulse)

Circuit and System Level Timing (continued)

- New Timing Components
  - t_p - clock period - The interval between occurrences of a specific clock edge in a periodic clock
  - $t_{pd,COMB}$ - total delay of combinational logic along the path from flip-flop output to flip-flop input
  - t_{slack} - extra time in the clock period in addition to the sum of the delays and setup time on a path
    - Can be either positive or negative
    - Must be greater than or equal to zero on all paths for correct operation

Circuit and System Level Timing (continued)

- Timing Equations

  $t_p = t_{slack} + (t_{pd,FF} + t_{pd,COMB} + t_s)$

  - For t_{slack} greater than or equal to zero,

    $t_p \geq \max\ (t_{pd,FF} + t_{pd,COMB} + t_s)$

    for all paths from flip-flop output to flip-flop input

- Can be calculated more precisely by using t_{PHL} and t_{PLH} instead of t_{pd} , but requires consideration of inversions through paths

Calculation of Allowable $t_{pd,COMB}$

- Compare the allowable combinational delay for a specific circuit:
  - a) Using edge-triggered flip-flops
  - b) Using master-slave flip-flops
- Parameters
  - $t_{pd,FF}(\max) = 1.0$ ns
  - $t_s(\max) = 0.3$ ns for edge-triggered flip-flops
  - $t_s = t_{wH} = 2.0$ ns for master-slave flip-flops
  - Clock frequency = 250 MHz

Calculation of Allowable $t_{pd,COMB}$ (continued)

- Calculations: t_p = 1/clock frequency = 4.0 ns
  - Edge-triggered: $4.0 \geq 1.0 + t_{pd,COMB} + 0.3$,
    $t_{pd,COMB} \leq 2.7$ ns
  - Master-slave: $4.0 \geq 1.0 + t_{pd,COMB} + 2.0$,
    $t_{pd,COMB} \leq 1.0$ ns
- Comparison: Suppose that for a gate, average $t_{pd} = 0.3$ ns
  - Edge-triggered: Approximately 9 gates allowed on a path
  - Master-slave: Approximately 3 gates allowed on a path

Overview

- Introduction of sequential circuits
- Basic storage elements
- Synchronous sequential logic analysis
- Synchronous sequential logic design
- Classic sequential logic circuits

The Design Procedure

- **Specification**
- **Formulation**
  - Obtain a state diagram or state table
- **State assignment**
  - Assign binary codes to the states
- **Flip-flop input equation determination**
  - Select flip-flop types and derive flip-flop equations from next state entries in the table
- **Output equation determination**
  - Derive output equations from output entries in the table
- **Optimization**
  - Optimize the equations
- **Technology mapping**
  - Find circuit from equations and map to flip-flops and gate technology
- **Verification**
  - Verify correctness of final design

Formulation: Finding a State Diagram

- A state is an abstraction of the history of the past applied inputs to the circuit (including power-up reset or system reset).
  - The interpretation of "past inputs" is tied to the synchronous operation of the circuit. E.g., an input value (other than an asynchronous reset) is measured only during the setup-hold time interval for an edge-triggered flip-flop.

- Examples:
  - State A represents the fact that a 1 input has occurred among the past inputs.
  - State B represents the fact that a 0 followed by a 1 have occurred as the most recent past two inputs.

Formulation: Finding a State Diagram (cont'd)

- In specifying a circuit, we use states to remember meaningful properties of past input sequences that are essential to predicting future output values.
- A sequence recognizer is a sequential circuit that produces a distinct output value whenever a prescribed pattern of input symbols occur in sequence, i.e., recognizes an input sequence occurrence.
- We will develop a procedure specific to sequence recognizers to convert a problem statement into a state diagram.
- Next, the state diagram, will be converted to a state table from which the circuit will be designed.

Sequence Recognizer Procedure

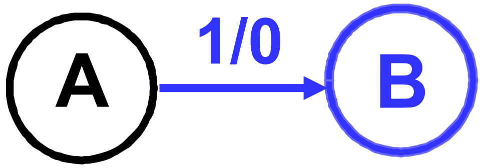
- To develop a sequence recognizer state diagram:
  - Begin in an initial state in which NONE of the initial portion of the sequence has occurred (typically “reset” state).
  - Add a state that recognizes that the first symbol has occurred.
  - Add states that recognize each successive symbol occurring.
  - The final state represents the input sequence (possibly less than the final input value) occurrence.
  - Add state transition arcs which specify what happens when a symbol ***NOT*** in the **proper sequence** has occurred.
  - Add other arcs on non-sequence inputs which transition to states that represent the input subsequence that has occurred.
- The last step is required because the circuit must recognize the input sequence ***regardless of where it occurs within the overall sequence applied since “reset”***.

Sequence Recognizer Procedure (cont'd)

- Example: Recognize the sequence 1101
  - Note that the sequence 1111101 contains 1101 and "11" is a proper sub-sequence of 1101.
- Thus, the sequential machine must remember that the first two one's have occurred as it receives another symbol.
- Also, the sequence 1101101 contains 1101 as both an initial subsequence and a final subsequence with some overlap, i. e., $\underline{1101}$101 or 110$\underline{1101}$.
- And, the 1 in the middle, 110$\underline{1}$101, is in both subsequences.
- The sequence 1101 must be recognized each time it occurs in the input sequence.

Example: Recognize 1101

- Define states for the sequence to be recognized:
  - assuming it starts with first symbol,
  - continues through each symbol in the sequence to be recognized, and
  - uses output 1 to mean the full sequence has occurred,
  - with output 0 otherwise.
- Starting in the initial state (Arbitrarily named "A"):
  - Add a state that recognizes the first "1."

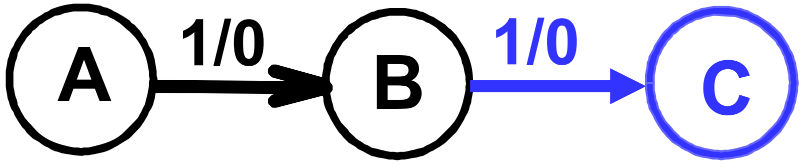


$$\begin{tikzcd} A \arrow[r, "1/0"] & B \end{tikzcd}$$

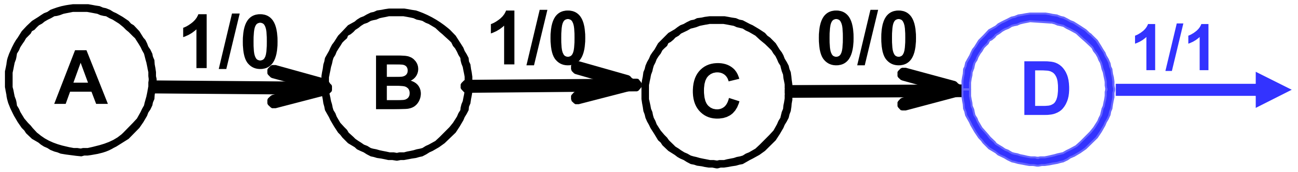
  - State "A" is the initial state, and state "B" is the state which represents the fact that the "first" one in the input subsequence has occurred. The output symbol "0" means that the full recognized sequence has not yet occurred.

Example: Recognize 1101 (cont'd)

- After one more 1, we have:
  - C is the state obtained when the input sequence has two "1"s.



- Finally, after 110 and a 1, we have:



- Transition arcs are used to denote the output function (**Mealy model**)
- Output 1 on the arc from D means the sequence has been recognized
- To what state should the arc from state D go? Remember: 110$\underline{1101}$?
- Note that D is the last state but the output 1 occurs for the input applied in D. This is the case when a ***Mealy model*** is assumed.

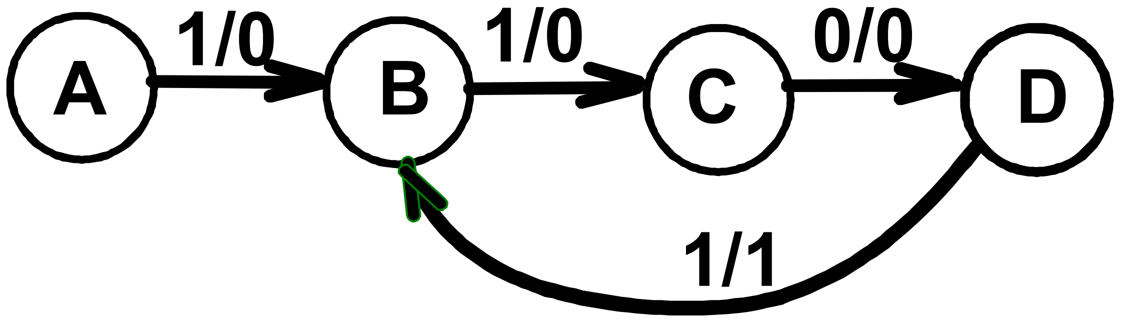
Example: Recognize 1101 (cont’d)

$$\begin{tikzcd}
A \arrow[r, "1/0"] & B \arrow[r, "1/0"] & C \arrow[r, "0/0"] & D \arrow[r, "1/1"] & {}
\end{tikzcd}$$

- Clearly the final 1 in the recognized sequence 1101 is a sub-sequence of 1101.
- It must be the first 1 in the sequence, since it cannot be preceded by a 1. Thus it should represent ***the same state reached from the initial state after a first 1 is observed***. We obtain:

$$\begin{tikzcd}
A \arrow[r, "1/0"] & B \arrow[r, "1/0"] & C \arrow[r, "0/0"] & D \arrow[ll, bend left, "1/1"]
\end{tikzcd}$$

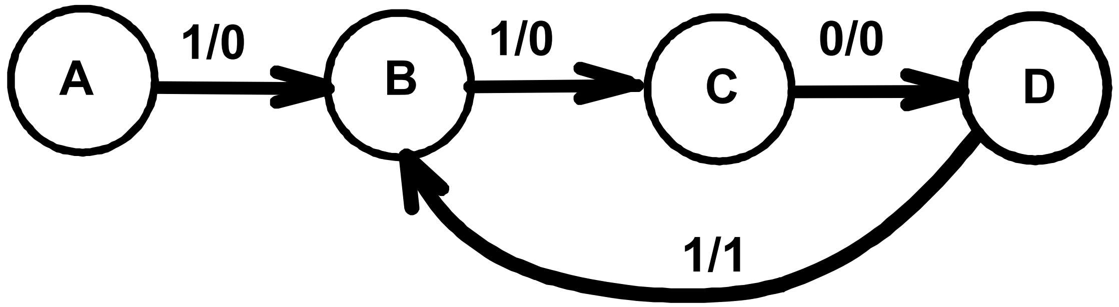
Example: Recognize 1101 (cont’d)



- The state have the following abstract meanings:
  - A: No proper sub-sequence of the sequence has occurred.
  - B: The sub-sequence 1 has occurred.
  - C: The sub-sequence 11 has occurred.
  - D: The sub-sequence 110 has occurred.
  - The 1/1 on the arc from D to B means that the last 1 has occurred and thus, the sequence is recognized.

Example: Recognize 1101 (cont'd)

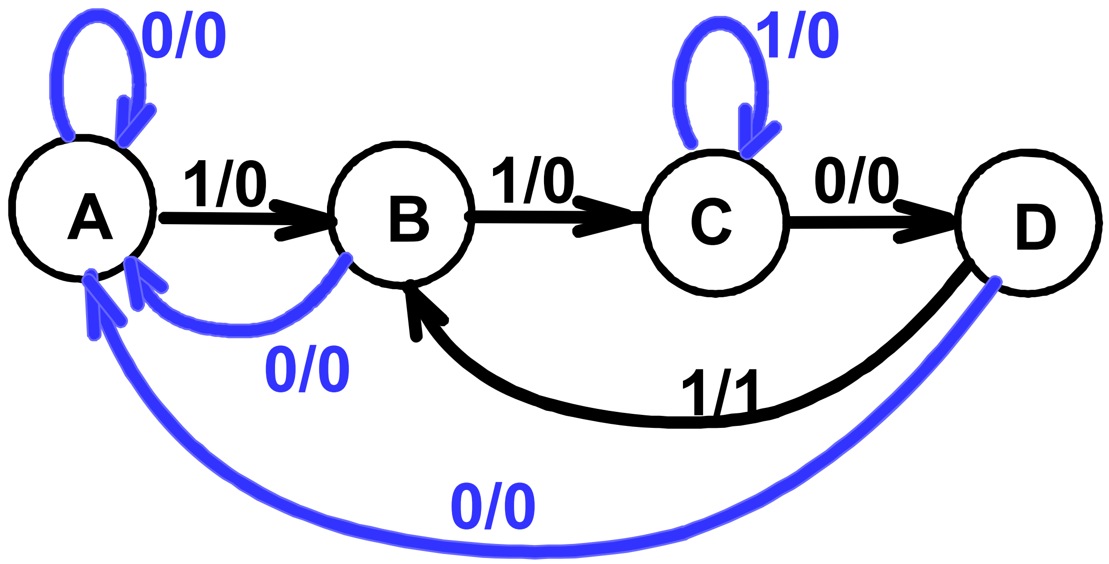
- The other arcs are added to each state for inputs not yet listed. Which arcs are missing?



- Answer:
- "0" arc from A
  "0" arc from B
  "1" arc from C
  "0" arc from D.

Example: Recognize 1101 (cont'd)

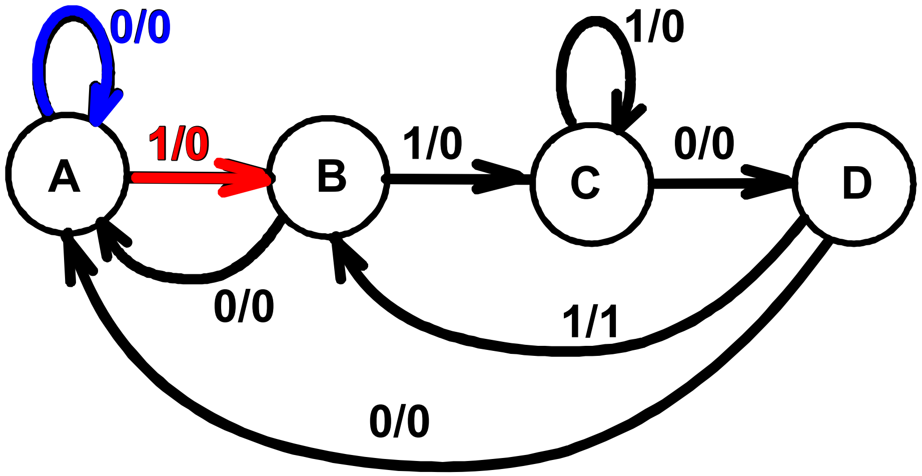
- State transition arcs must represent the fact that an input subsequence has occurred. Thus, we get:



- Note that the 1 arc from state C to state C implies that State C means ***two or more 1's have occurred***.

Formulation: Find State Table

- From the state diagram, we can fill in the state table.
- There are 4 states, one input, and one output. We will choose the form with four rows, one for each current state.
- From State A, the 0 and 1 input transitions have been filled in along with the outputs.



Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

Note that state table can also be used to merge equivalent states.

What would the state diagram and state table look like for the Moore model?

Example: Moore Model for Sequence 1101

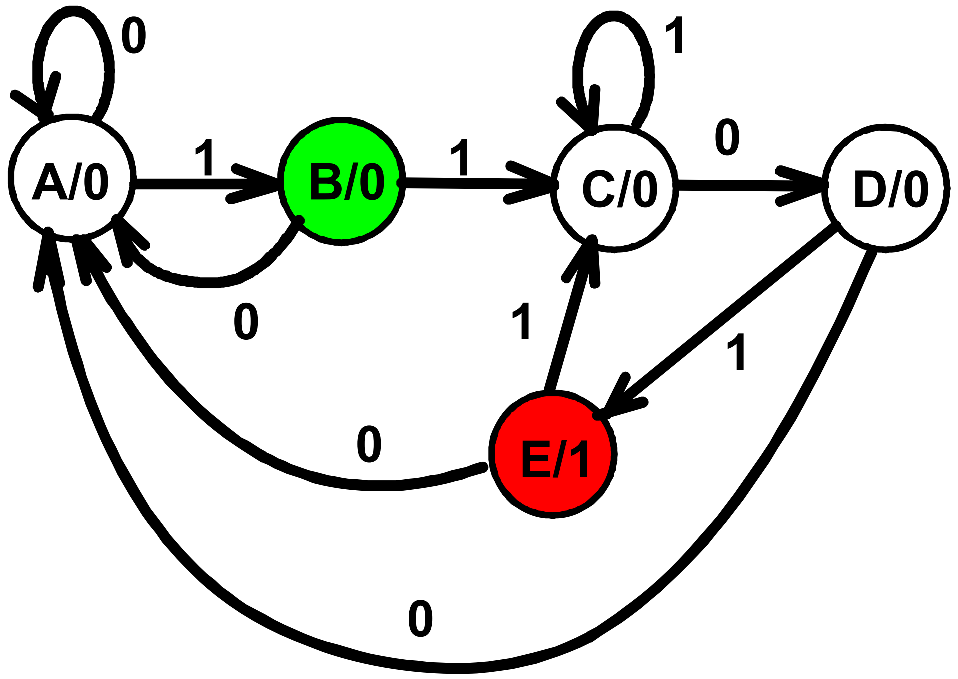
- For the Moore model, outputs are associated with states.

- We need to add a state "E" with output value 1 for the final 1 in the recognized input sequence.
  - This new state E, though similar to B, would generate an output of 1 and thus be different from B.

- The Moore model for a sequence recognizer usually has more states than the Mealy model.

Example: Moore Model (cont'd)

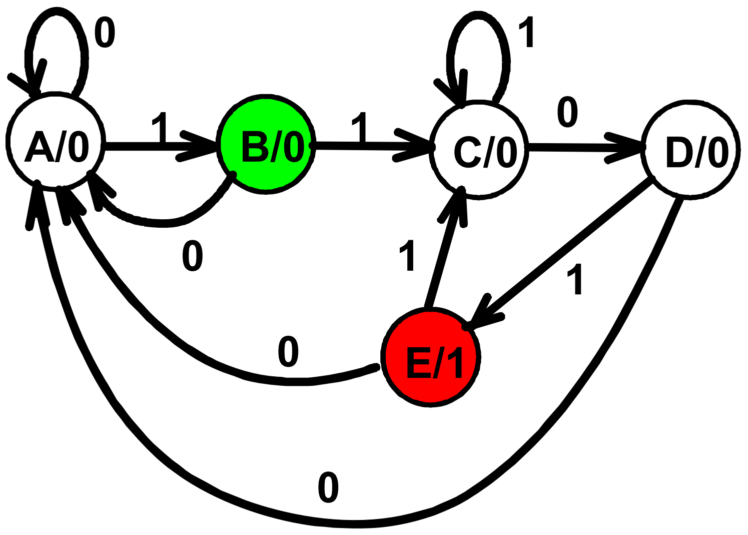
- We mark outputs on states for Moore model
- Arcs now show only state transitions
- Add a new state E to produce the output 1



- Note that the new state, E produces the same behavior in the future as state B. But it gives a different output at the present time. Thus, these states do represent a ***different abstraction*** of the input history.

Example: Moore Model (cont'd)

- Fill in the state table according to the stable diagram
- Memory aid re more state in the Moore model: "Moore is More."



Present State	Next State x=0	Next State x=1	Output y
A	A	B	0
B	A	C	0
C	D	C	0
D	A	E	0
E	A	C	1

State Assignment

- Each of the ***m*** states must be assigned a unique code
- Minimum number of bits required is ***n*** such that

  $$n \geq \lceil \log_2 m \rceil$$

  where $\lceil x \rceil$ is the smallest integer $\geq x$
- There are useful state assignments that use more than the minimum number of bits
- There are 2^n - m unused states

State Assignment: Example 1

Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
A	A	B	0	0
B	A	B	0	1

- How many assignments of codes with a minimum number of bits?
  - Two: A = 0, B = 1 or A = 1, B = 0
- Does it make a difference?
  - Only in variable inversion, so small, if any.

State Assignment: Example 2

Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

- How many assignments of codes with a minimum number of bits?
- Does code assignment make a difference in cost?

State Assignment: Example 2 (cont'd)

- Counting Order Assignment: A = 00, B = 01, C = 10, D = 11
- The resulting coded state table:

Present State $Y_1\ Y_2$	Next State $x = 0$ $D_1\ D_2$	Next State $x = 1$ $D_1\ D_2$	Output $x = 0$	Output $x = 1$
0 0	0 0	0 1	0	0
0 1	0 0	1 0	0	0
1 0	1 1	1 0	0	0
1 1	0 0	0 1	0	1

State Assignment: Example 2 (cont’d)

- Gray Code Assignment: A = 00, B = 01, C = 11, D = 10
- The resulting coded state table:

Present State $Y_1 Y_2$	Next State x = 0 $D_1 D_2$	x = 1 $D_1 D_2$	Output x = 0	x = 1
0 0	0 0	0 1	0	0
0 1	0 0	1 1	0	0
1 1	1 0	1 1	0	0
1 0	0 0	0 1	0	1

Find Flip-Flop Input and Output Equations: Counting Order Assignment

- Assume D flip-flops
- Interchange the bottom two rows of the state table, to obtain K-maps for D1, D2, and Z:

D_1

		X	
	0	0	
	0	1	Y_2
Y_1	0	0	Y_2
Y_1	1	1	

D_2

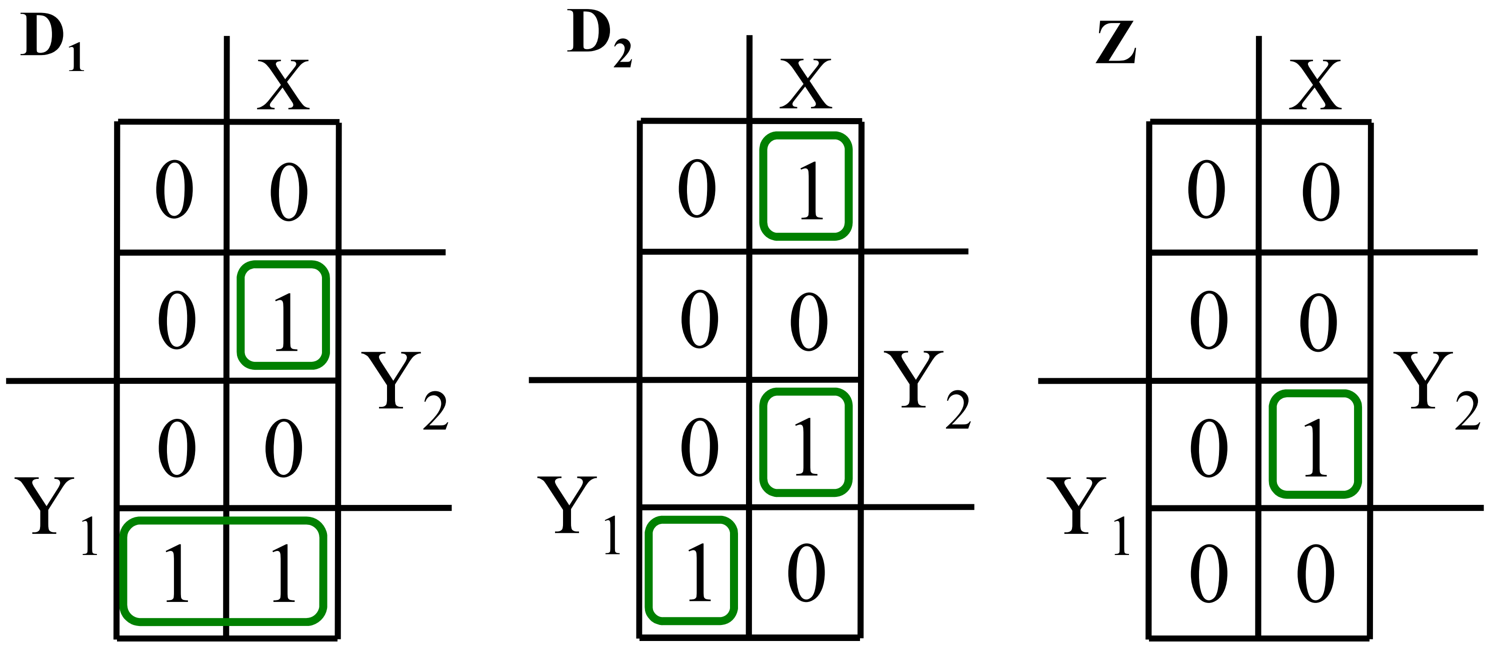
		X	
	0	1	
	0	0	Y_2
Y_1	0	1	Y_2
Y_1	1	0	

Z

		X	
	0	0	
	0	0	Y_2
Y_1	0	1	Y_2
Y_1	0	0	

Optimization: Counting Order Assignment

- Performing two-level optimization:

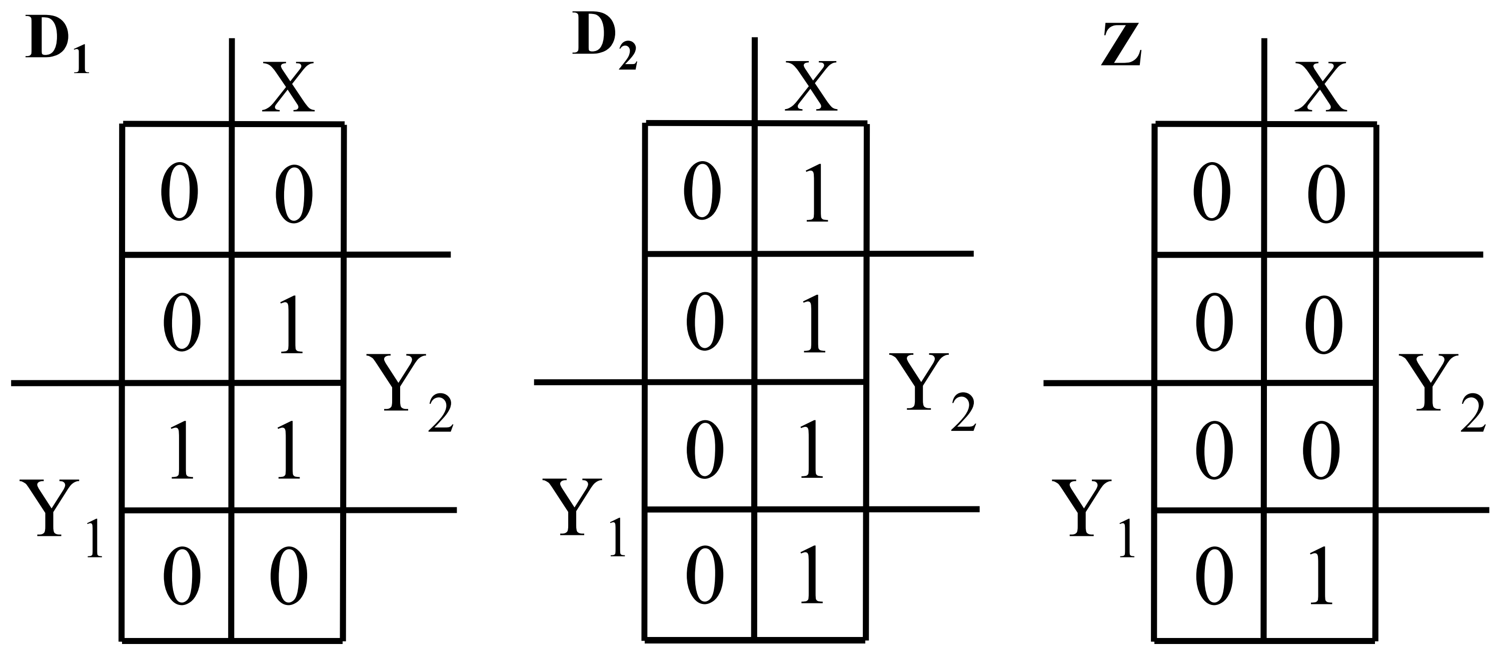


- $D_1 = Y_1\overline{Y_2} + X\overline{Y_1}Y_2$
  $D_2 = \overline{X}Y_1\overline{Y_2} + X\overline{Y_1}\,\overline{Y_2} + XY_1Y_2$
  $Z \;\; = XY_1Y_2$

Gate Input Cost = ?

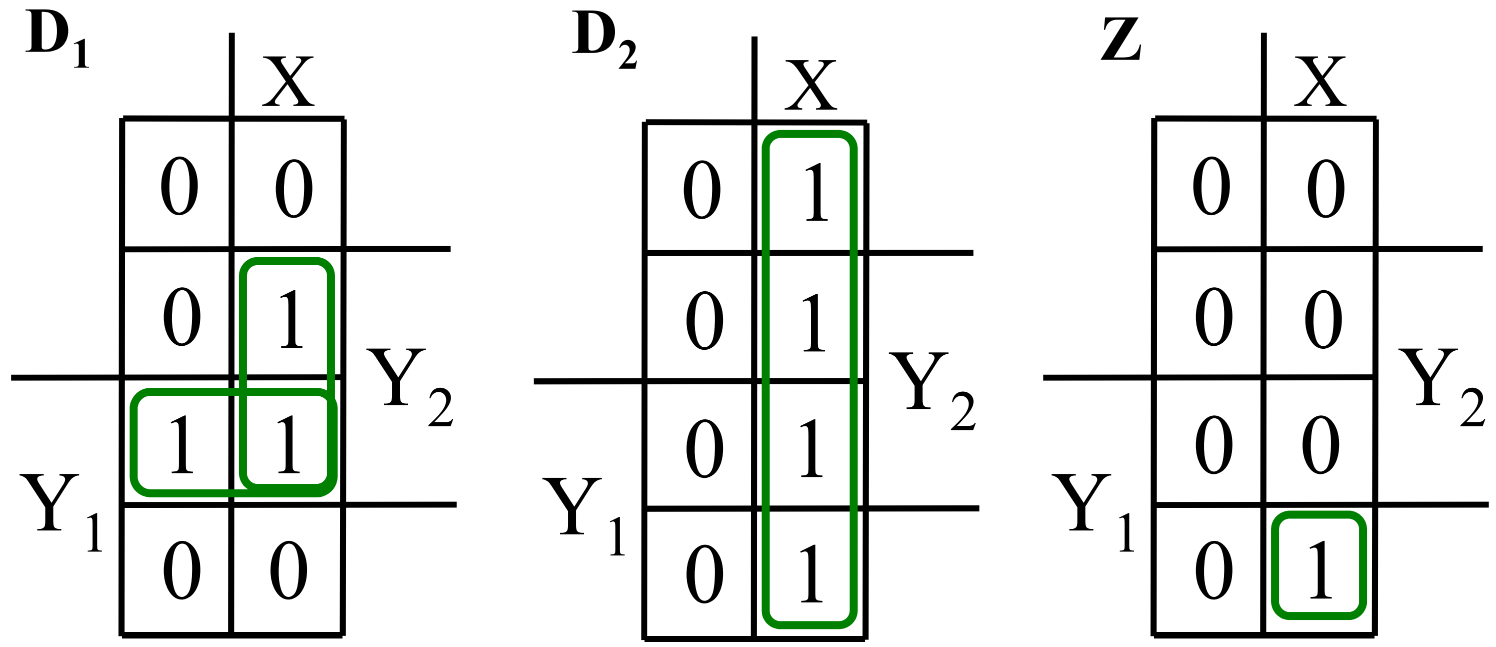
Find Flip-Flop Input and Output Equations: Gray Code Assignment

- Assume D flip-flops
- Obtain K-maps for D1, D2, and Z:



Optimization: Gray Code Assignment

- Performing two-level optimization:

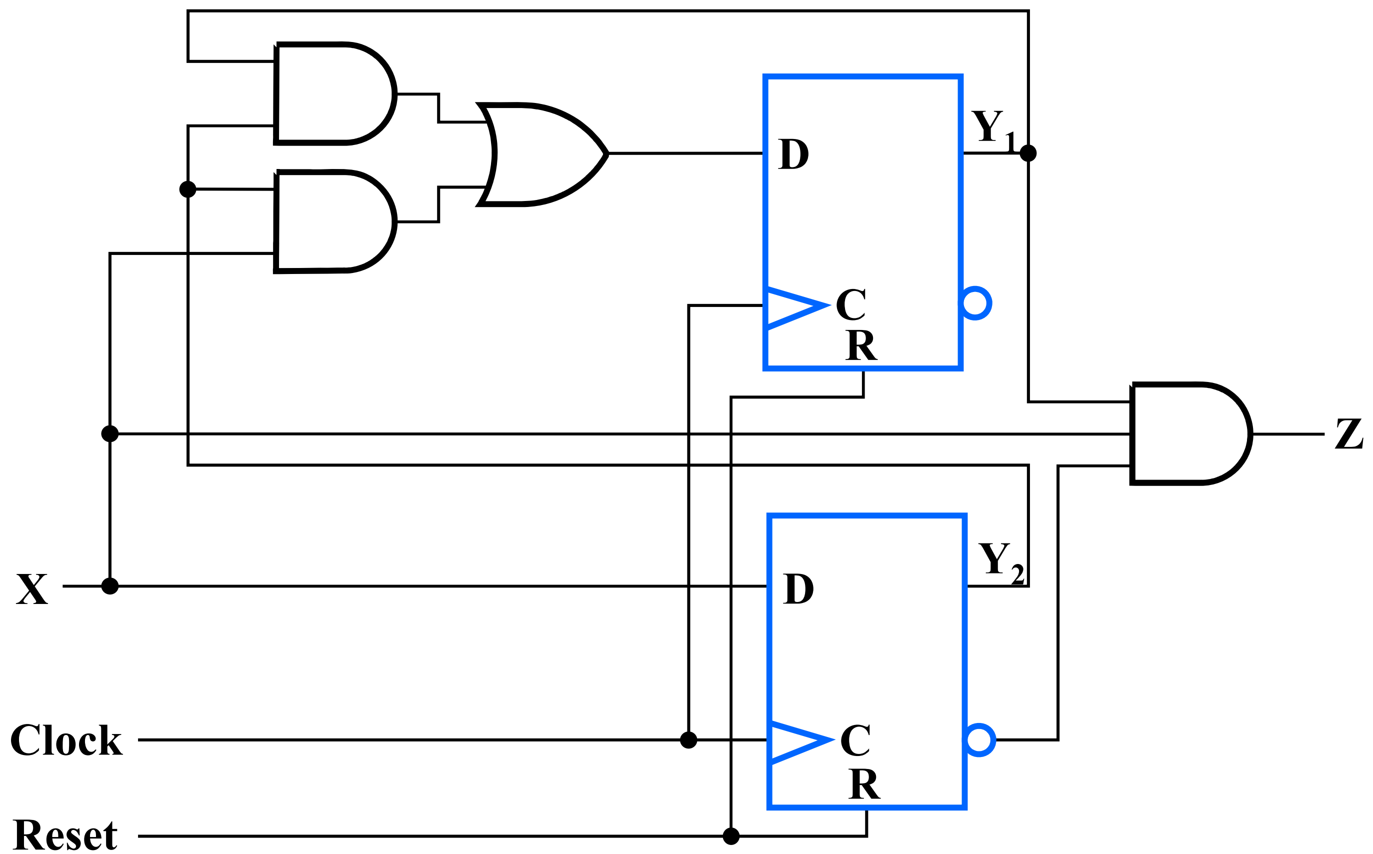


- $D_1 = Y_1Y_2 + XY_2$
  $D_2 = X$
  $Z = XY_1\overline{Y}_2$

Gate Input Cost = ?

Map Technology: Gray Code Assignment

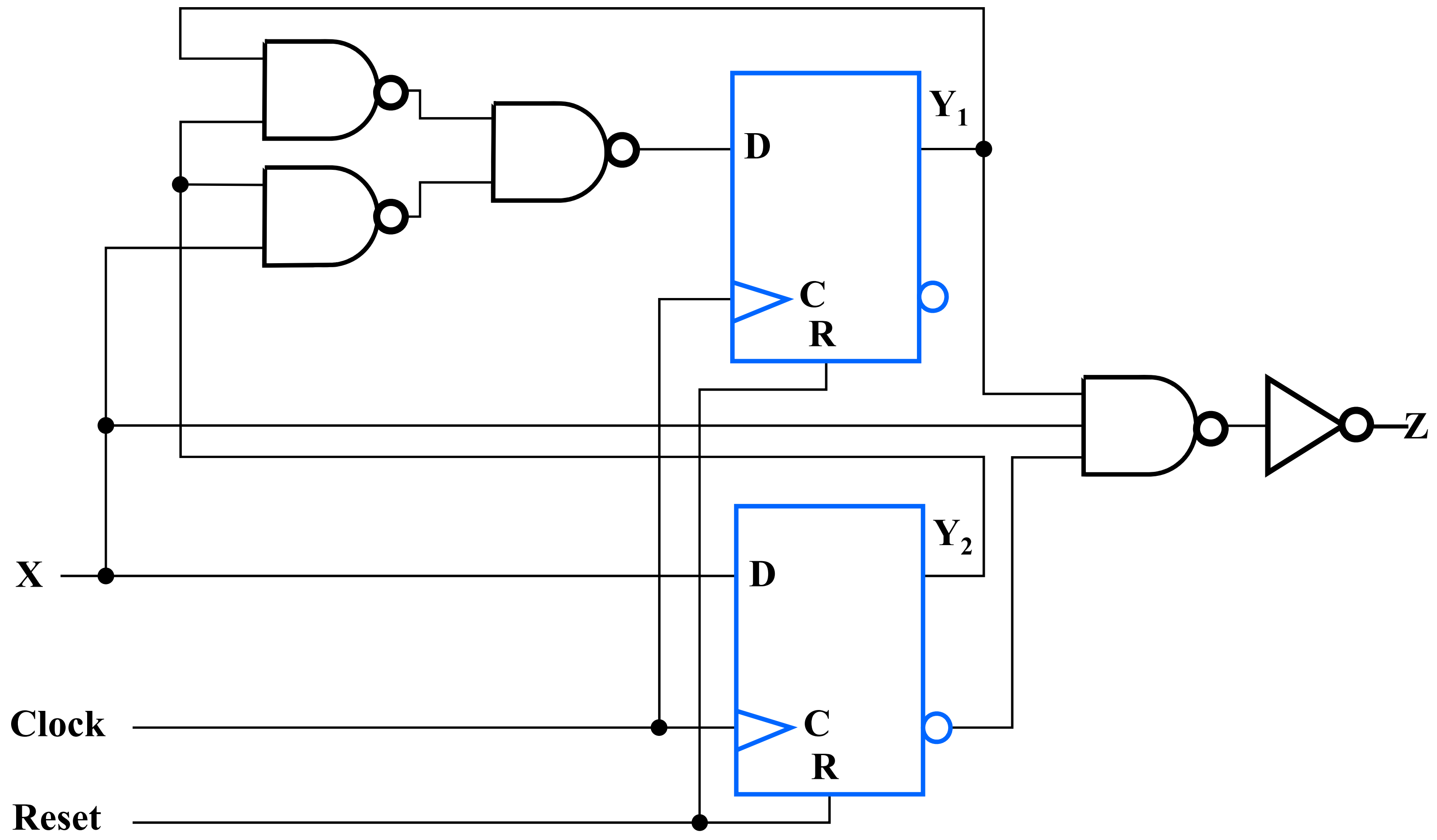
- D Flip-flops with Reset (not inverted)



Initial circuit

Mapped Circuit

- NAND gates with up to 4 inputs and inverters



Final circuit

State Assignment

- Generally different assignments will result in different output functions, excitation functions, and the complexity of circuit varies as well. The tasks of state assignment are:
  - Determine the length of code
  - Find the best state assignment or close to the best
- Finding a best assignment can be extremely hard when n is large enough. And the performance of the state assignment also depends on the type of triggers. In practice, state assignment is processed in an engineering way, where the following rules are applied.

Basic Rules For State Assignment

- The states which have the same next states with the same inputs should be assigned adjacent binary codes
- The next states of a present state with adjacent inputs should be assigned adjacent binary codes
- The states with the same output should be assigned adjacent binary codes
- Initial state or the most frequently used state should be assigned code 0

Basic Rules For State Assignment (cont'd)

- The first rule is the most important, should be considered first
- According to the former three rules, state pairs with more present times should be assigned adjacent codes with high priority

Example: State Assignment

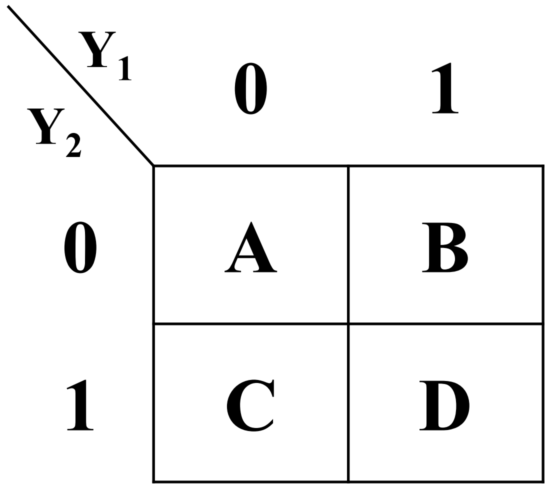
Present State	Next State/ Output	
	X=0	X=1
A	C/0	D/0
B	C/0	A/0
C	B/0	D/0
D	A/1	B/1

- Rule 1: A-B, A-C
- Rule 2: C-D, A-C, B-D, A-B
- Rule 3: A-B, A-C, B-C
- Rule 4: A = 0

Example: State Assignment (cont'd)

- n=2, since 2 triggers is enough to represent 4 states
- Determine the assignment
  - Rule 1: A-B, A-C
  - Rule 2: C-D, A-C, B-D, A-B
  - Rule 3: A-B, A-C, B-C
  - Rule 4: A = 0

Example: State Assignment (cont'd)



	Y_2	Y_1
A	0	0
B	0	1
C	1	0
D	1	1

- Rule 1: A-B, A-C
- Rule 2: C-D, A-C, B-D, A-B
- Rule 3: A-B, A-C, B-C
- Rule 4: A = 0

Example: State Assignment (cont’d)

- Final State Table

Present State	Next State/ Output	
	X=0	X=1
A	C/0	D/0
B	C/0	A/0
C	B/0	D/0
D	A/1	B/1

Present State	$y_2^{(t+1)}y_1^{(t+1)}$ / Output	
y_2 y_1	X=0	X=1
0 0	10/0	11/0
0 1	10/0	00/0
1 1	00/1	01/1
1 0	01/0	11/0

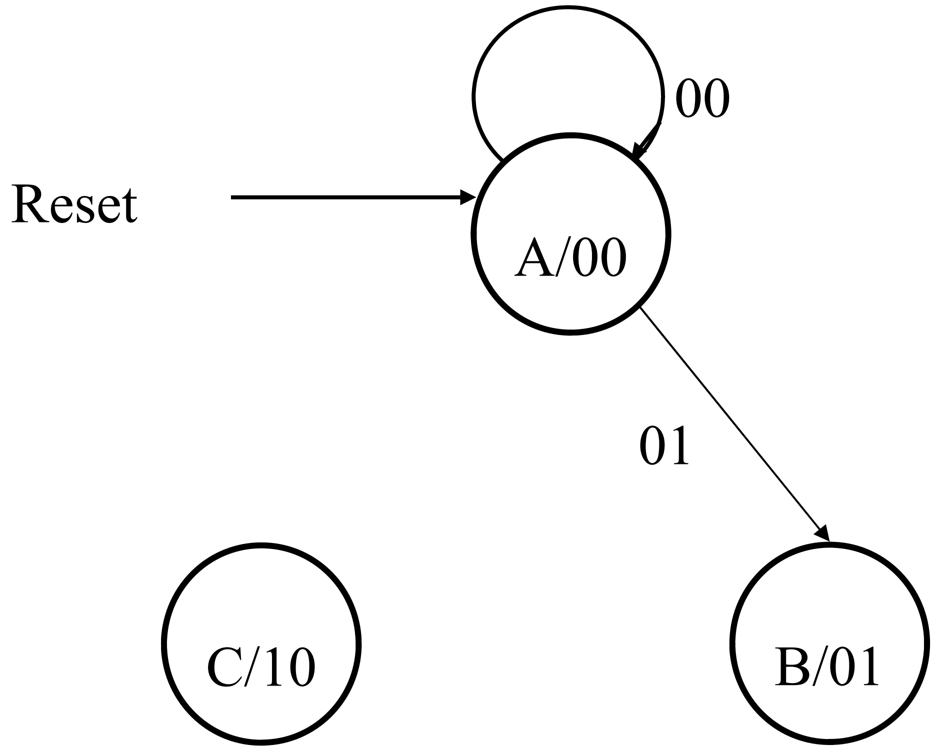
- Sometimes, the assignment that fit the requirement is not unique. And any of them is suitable.

Sequential Design: Example 3

- Design a sequential modulo 3 accumulator for 2-bit operands
- Definitions:
  - Modulo *n* adder - an adder that gives the result of the addition as the remainder of the sum divided by *n*
    - Example: 2 + 2 modulo 3 = remainder of 4/3 = 1
  - Accumulator - a circuit that "accumulates" the sum of its input operands over time - it adds each input operand to the stored sum, which is initially 0.
- Stored sum: (Y_1,Y_0), Input: (X_1,X_0), Output: (Z_1,Z_0)

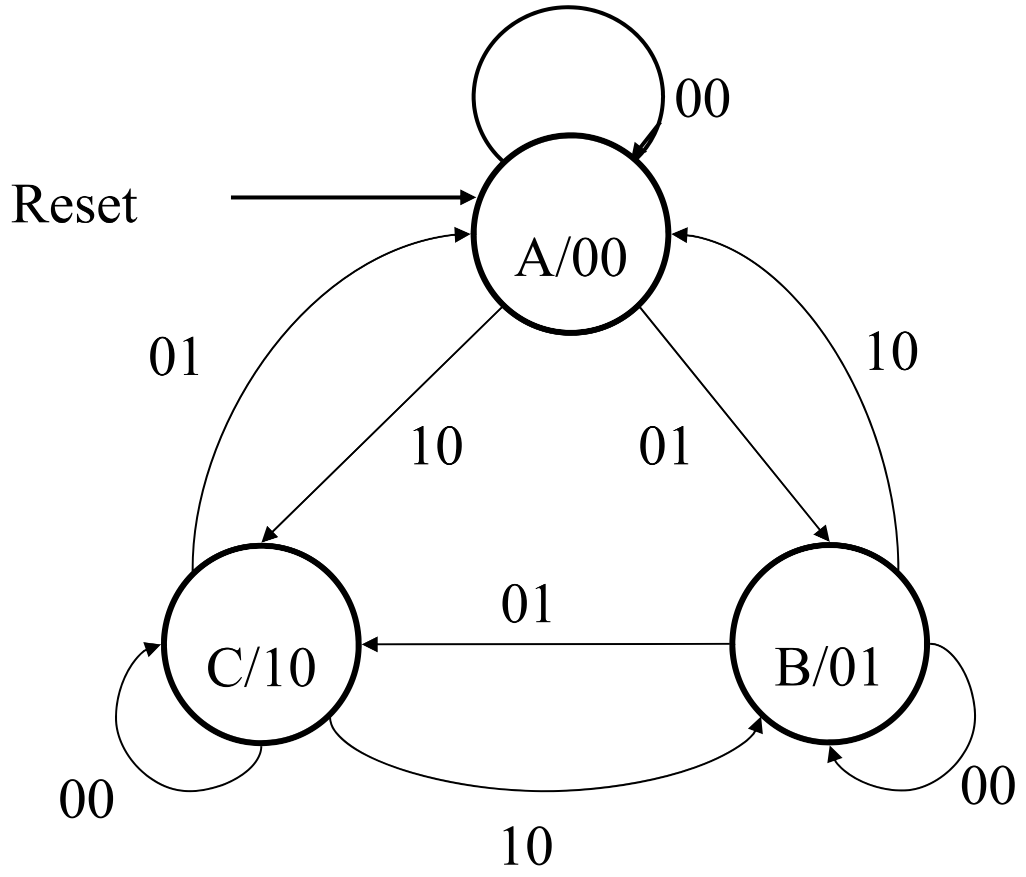
Example 3 (continued)

- Complete the state diagram:



Example 3 (continued)

- Complete the state diagram:



Example 3 (continued)

- Complete the state table

Y1Y0 \ X1X0	00	01	11	10	Z_1Z_0
	$Y_1(t+1)$, $Y_0(t+1)$	$Y_1(t+1)$, $Y_0(t+1)$	$Y_1(t+1)$, $Y_0(t+1)$	$Y_1(t+1)$, $Y_0(t+1)$	
A (00)	00		X		00
B (01)			X		01
- (11)	X	X	X	X	11
C (10)			X		10

- State Assignment: $(Y_1,Y_0) = (Z_1,Z_0)$
- Codes are in gray code order to ease use of K-maps in the next step

Example 3 (continued)

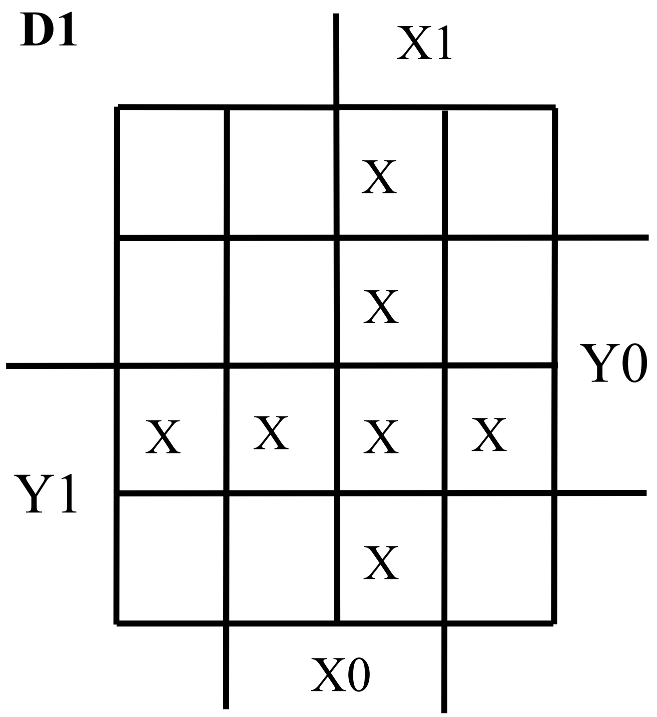
- Complete the state table

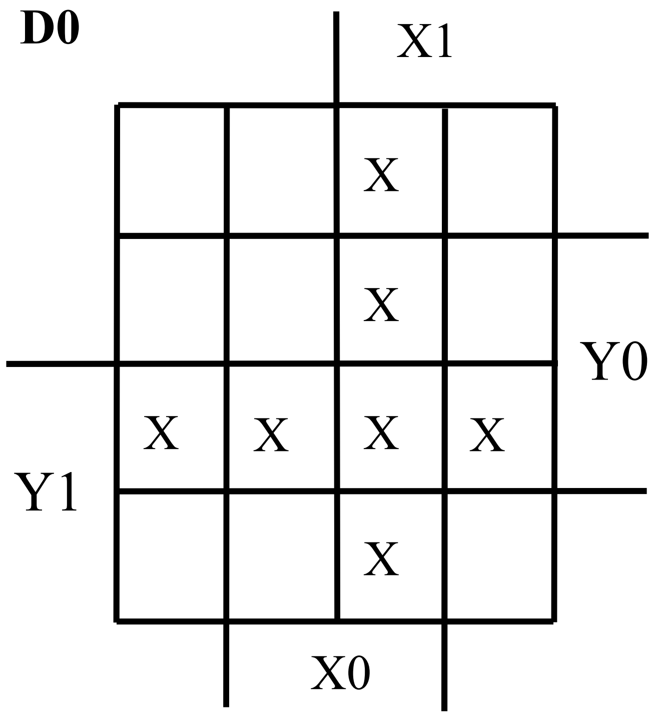
Y1Y0 \ X1X0	00	01	11	10	Z_1Z_0
	$Y_1(t+1)$, $Y_0(t+1)$	$Y_1(t+1)$, $Y_0(t+1)$	$Y_1(t+1)$, $Y_0(t+1)$	$Y_1(t+1)$, $Y_0(t+1)$	
A (00)	00	01	X	10	00
B (01)	01	10	X	00	01
- (11)	X	X	X	X	11
C (10)	10	00	X	01	10

- State Assignment: $(Y_1,Y_0) = (Z_1,Z_0)$
- Codes are in gray code order to ease use of K-maps in the next step

Example 3 (continued)

- Find optimized flip-flop input equations for D flip-flops





- $D_1 =$
- $D_0 =$

Example 3 (continued)

- Find optimized flip-flop input equations for D flip-flops

D1

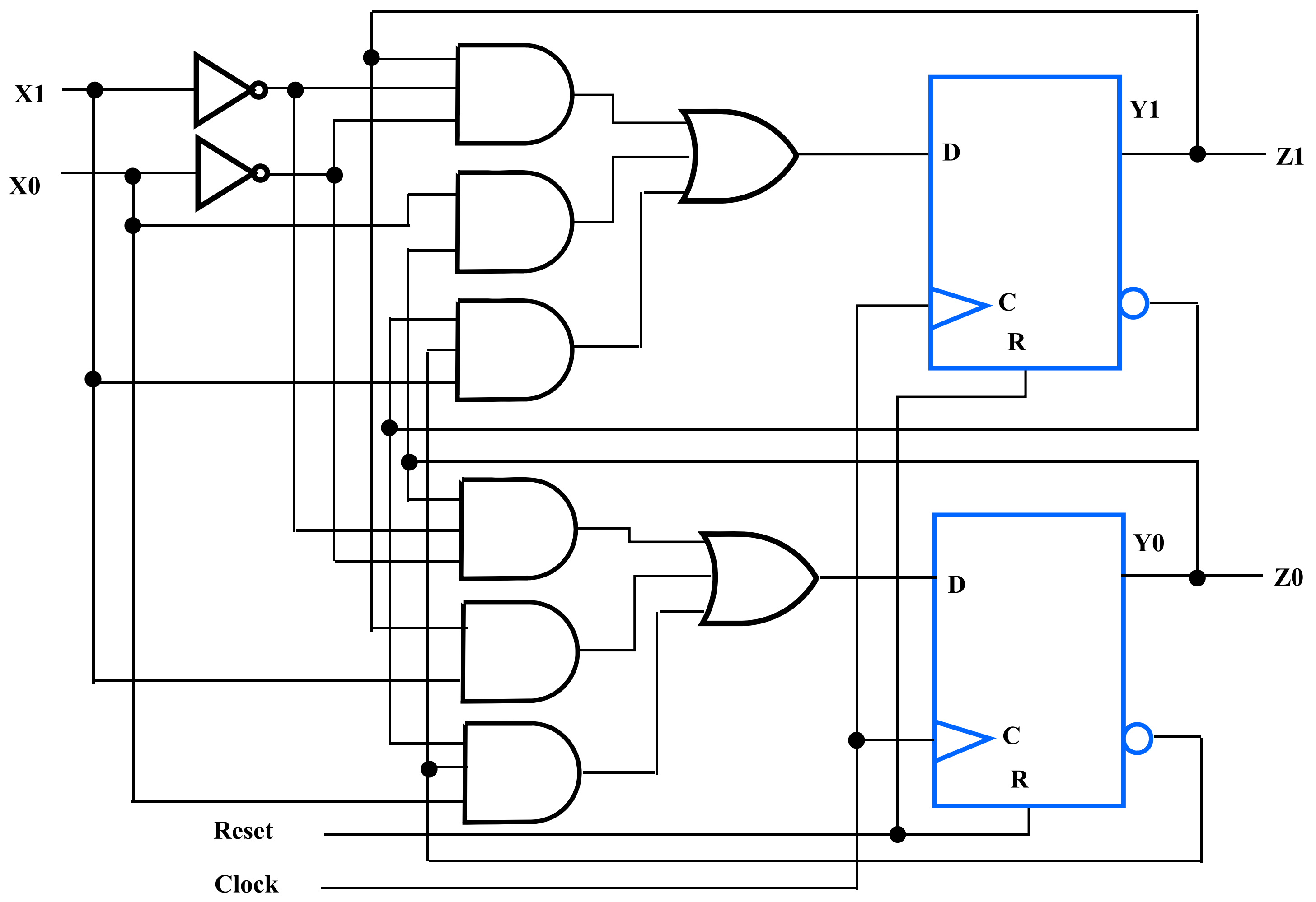
				X1	X1
			X0	X0	
		0	0	X	1
Y0		0	1	X	0
Y0	Y1	X	X	X	X
	Y1	1	0	X	0

D0

				X1	X1
			X0	X0	
		0	1	X	0
Y0		1	0	X	0
Y0	Y1	X	X	X	X
	Y1	0	0	X	1

- $D_1 = Y_1\overline{X_1}\,\overline{X_0} + Y_0X_0 + \overline{Y_1}\,\overline{Y_0}X_1$
- $D_0 = Y_0\overline{X_1}\,\overline{X_0} + Y_1X_1 + \overline{Y_1}\,\overline{Y_0}X_0$

Circuit - Final Result with AND, OR, NOT



Unused States: The State Assignment Problem

- The previous example has only 5 states, represented by 3-bit values. So, we have three unused state assignments in this case.

- This is no real problem, except that at power up, the machine may start up in one of the unused states. Or, it could get into an unused state due to electrical noise (lightning, bad connections, etc).

- One possibility is to add **RESET** logic to force the machine into state 000 on power up, or on detect of unused state. If entry to an unused state is considered fatal, then stop the FSM (force it into an ERROR state, where it stays) and transmit an error code.

Unused States: In the hardware Implementation of the FSM

- A nasty problem is that of the machine appearing to **freeze up** if it gets into an unused state. This can happen if the unused states point to themselves or each other.
- If all unused states point to used states, the machine is said to be "**self starting**" and, after power up, will function correctly. If the machine is self-starting, entering an unused state will cause a brief erratic operation, then the machine will resume correct operation.
- So, if the application is fault-tolerant, and the machine is self-starting, we are OK to ignore the problem beyond verifying self start capability.
- We can guarantee self start (assuming state 000 is a state we are using) by replacing the Xs in the implementation diagram above with 0s. This will force the machine into state 000 whenever an unused state is entered. The problem, of course, is more complex steering logic.

Overview

- Introduction of sequential circuits
- Basic storage elements
- Synchronous sequential logic analysis
- Synchronous sequential logic design
- Classic sequential logic circuits

Classic Sequential Logic Elements

- Registers
  - Basic introduction
  - Registers in the digital system
  - Shift Registers

- Counters
  - Basic introduction
  - Ripple Counters
  - Synchronous Counters

Classic Sequential Logic Elements

- Registers
  - Basic introduction
  - Registers in the digital system
  - Shift Registers

- Counters
  - Basic introduction
  - Ripple Counters
  - Synchronous Counters

Registers

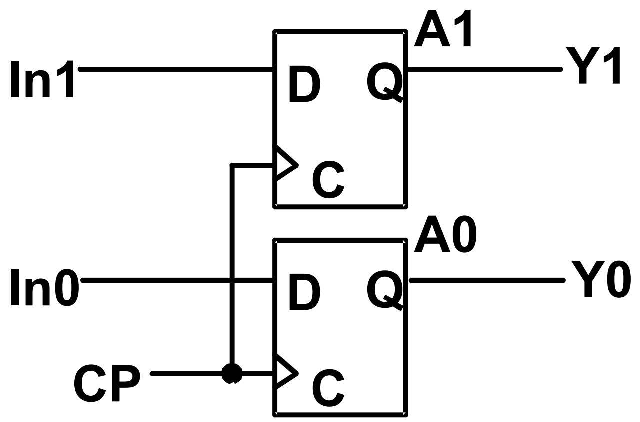
- A collection of binary storage elements
  - A register is a sequential logic which can be defined by a state table
  - More often, think of a register as storing a vector of binary values
  - Frequently used to perform simple data storage and data movement and processing operations
- Common sequential devices
  - They're a good example of sequential analysis and design
  - They are also frequently used in building larger sequential circuits
- Hold larger quantities of data than individual flip-flops
  - Registers are central to the design of modern processors
  - There are many kinds of registers to serve different purpose

Registers vs. Flip-Flops

- Flip-flops are limited because they can store only one bit
  - Two flip-flops are used for two-bit registers.
  - Most computers work with integers and single-precision floating-point numbers that are 32-bits long.
- While a register is an extension of a flip-flop that can store multiple bits
- Registers are commonly used as temporary storage in a processor
  - They are faster and more convenient than main memory.
  - More registers can help speed up complex calculations.

Example: 2-Bit Register

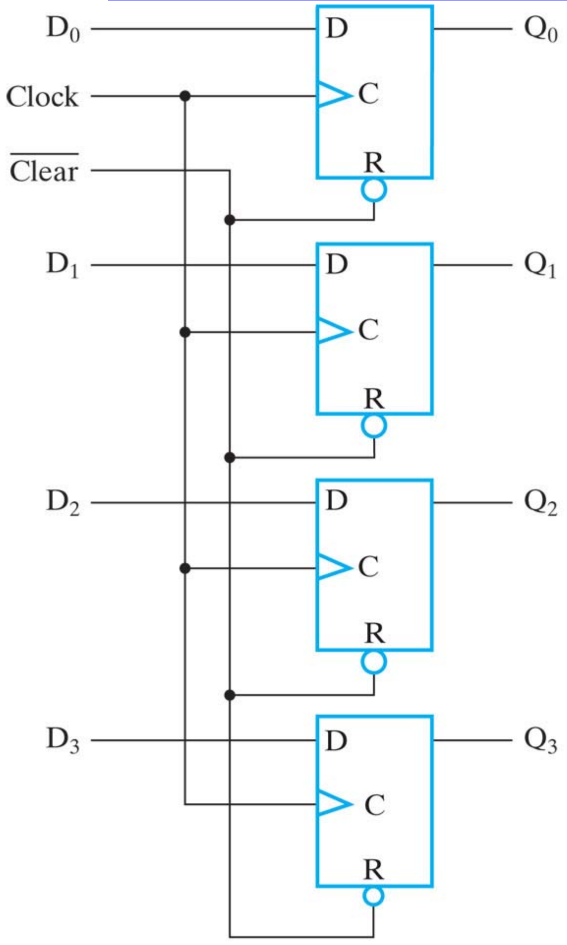
- How many states are there?
- How many input combinations? Output combinations?
- What is the output function?
- What is the next state function?
- Mealy or Moore?



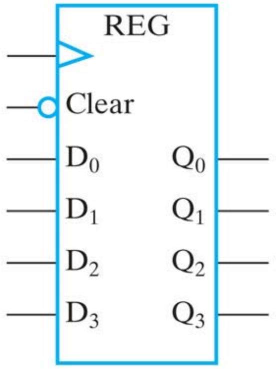
Current State	Next State A1(t+1) A0(t+1) For In1 In0 =	Output (=A1 A0)
A1 A0	00 01 10 11	Y1 Y0
0 0	00 01 10 11	0 0
0 1	00 01 10 11	0 1
1 0	00 01 10 11	1 0
1 1	00 01 10 11	1 1

- What are the quantities above for an n-bit register?

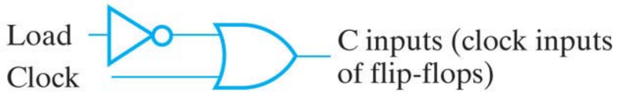
Example of 4-Bit Register: Clock Gating



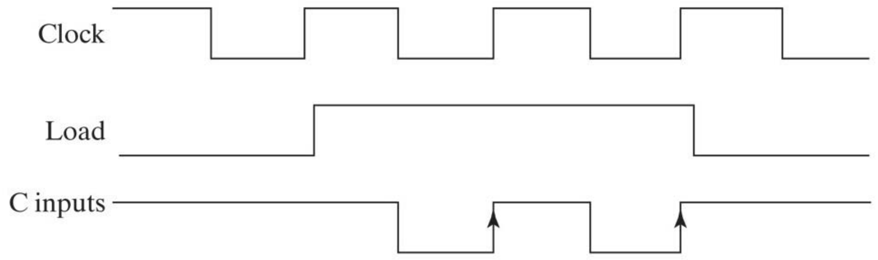
(a) Logic diagram



(b) Symbol

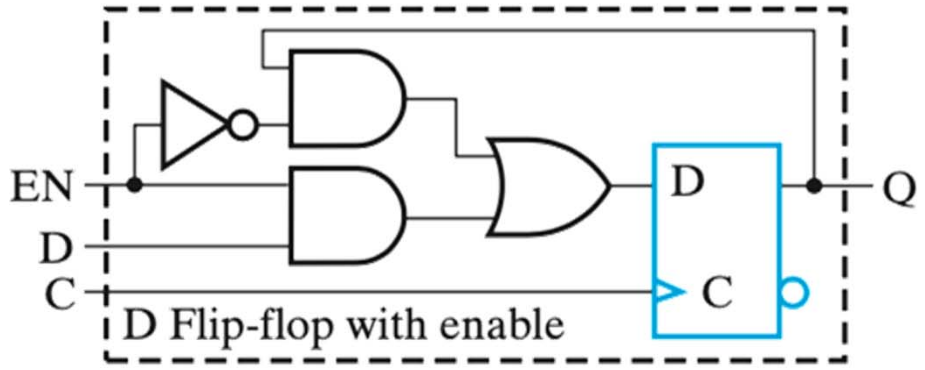


(c) Load control input

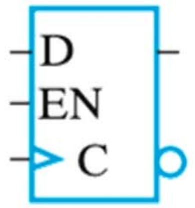


(d) Timing diagram

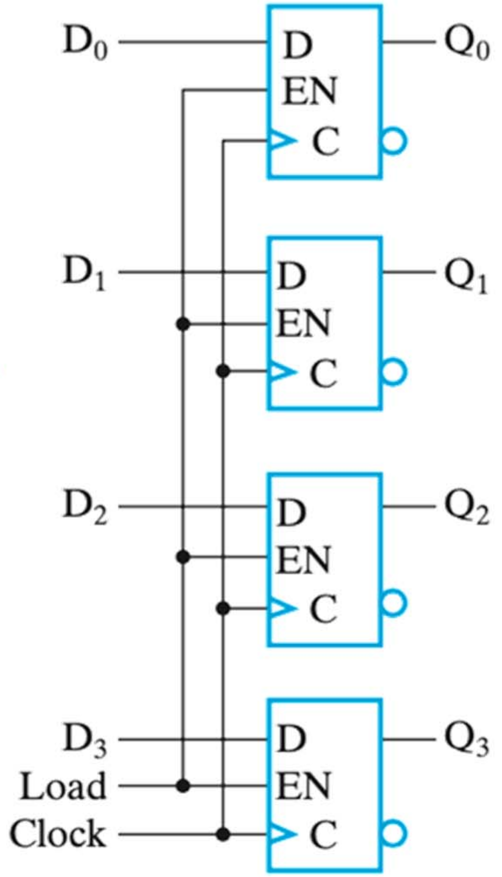
Flip-Flops With EN



(a)



(b)



(c)

Register Transfers

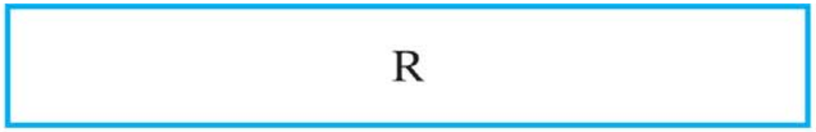
- The registers are assumed to be basic components of the digital system

- The movement of the data stored in registers and the processing performed on the data are referred to as ***register transfer operations***, which are specified by the following three basic components:
  - The set of registers in the system
  - The operations that are performed on the data stored in the registers
  - The control that supervises the sequence of operations in the system

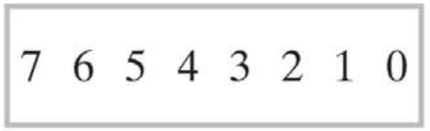
Microoperation

- A register has the capability to perform one or more elementary operations such as *load*, *count*, *add*, *subtract*, and *shift*
- An elementary operation performed on data stored in registers is called a ***microoperation***, e.g.,
  - Loading the contents of one register into another
  - Adding the contents of two registers
  - Incrementing the contents of a register
- Four types of most widely used microoperations
  - *Transfer*, *Arithmetic*, *Logic* and *Shift*

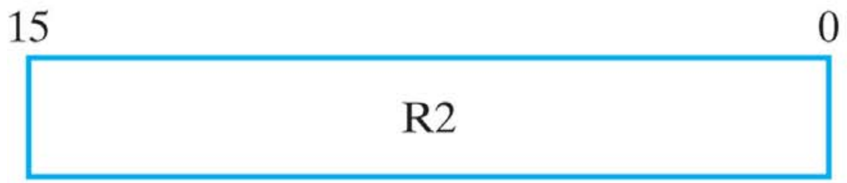
Register Representation



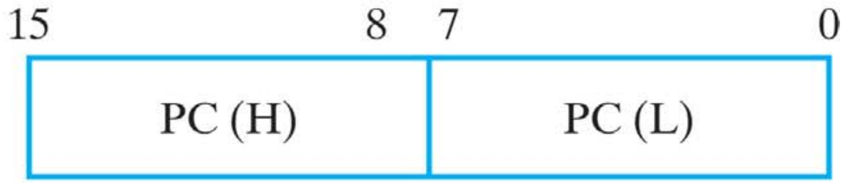
(a) Register R



(b) Individual bits of 8-bit register

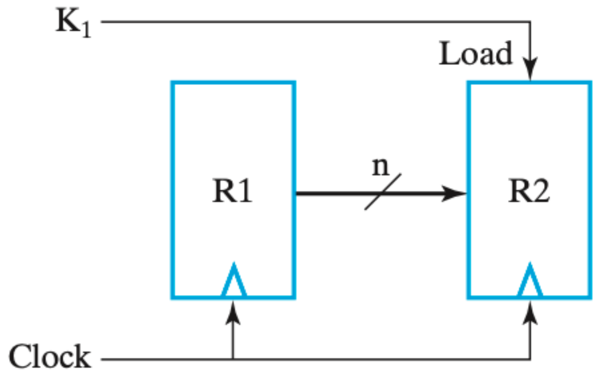


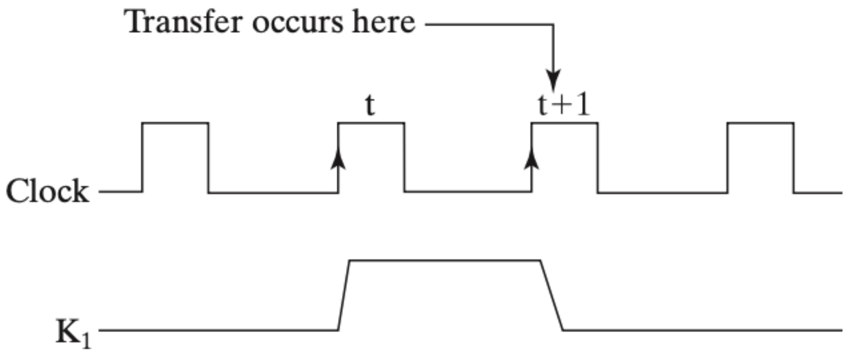
(c) Numbering of 16-bit register



(d) Two-part 16-bit register

$$\text{if } (K_1 = 1) \text{ then } (R2 \leftarrow R1)$$





Microoperations

- Logical Groupings:
  - Transfer - move data from one set of registers to another
  - Arithmetic - perform arithmetic on data in registers
  - Logic - manipulate data or use bitwise logical operations
  - Shift - shift data in registers

Arithmetic operations	Logical operations
$+$ Addition	$\vee$ Logical OR
$-$ Subtraction	$\wedge$ Logical AND
$*$ Multiplication	$\oplus$ Logical Exclusive OR
$/$ Division	$\overline{}$ Not

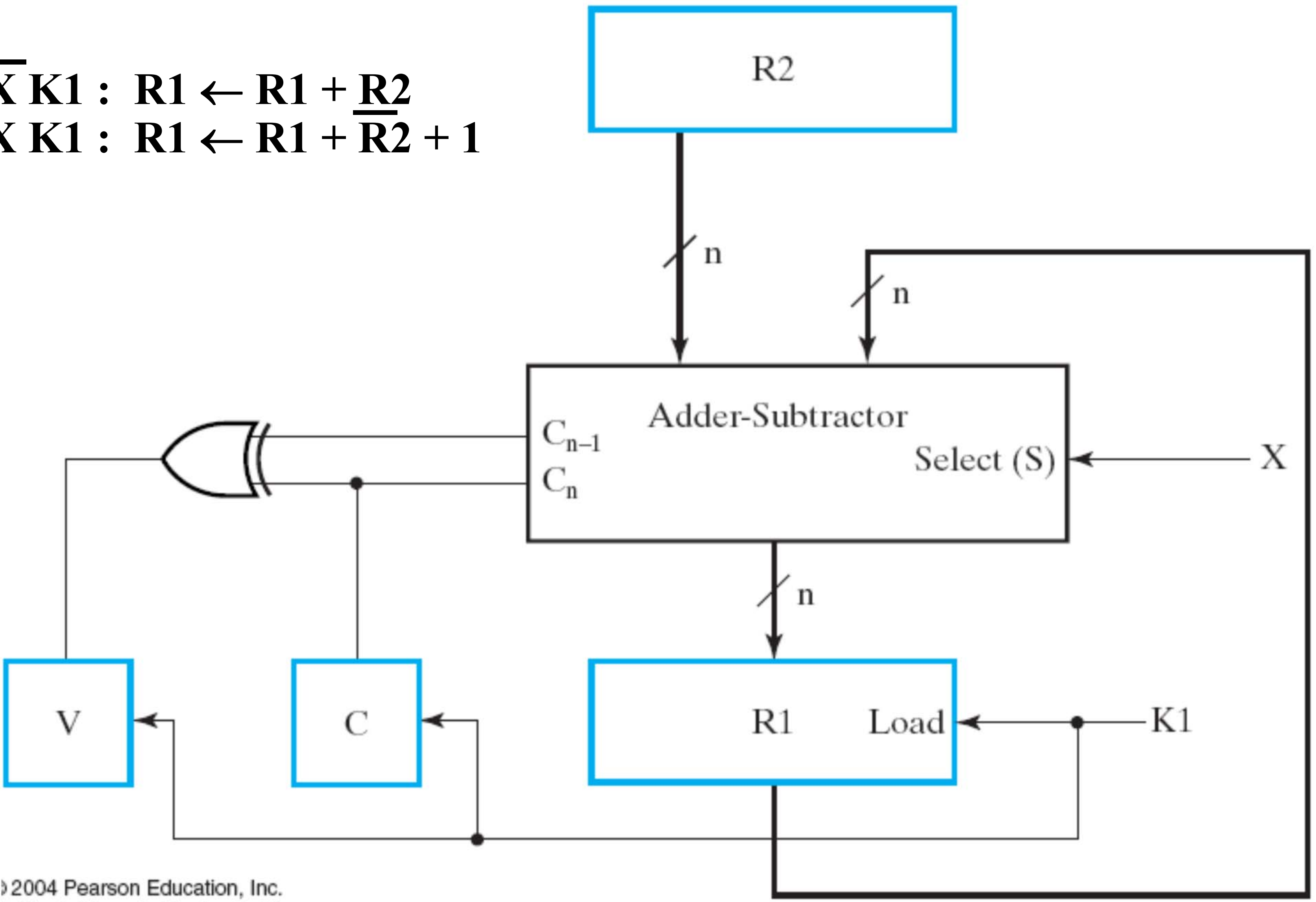
Control Expressions

- The <u>control expression</u> for an operation appears to the left of the operation and is separated from it by a colon
- Control expressions specify the <u>logical condition</u> for the operation to occur
- Control expression values of:
  - Logic "1" -- the operation occurs.
  - Logic "0" -- the operation does not occur.

- Example:
  $\overline{X}\,K1 : R1 \leftarrow R1 + R2$
  $X\,K1 : R1 \leftarrow R1 + \overline{R2} + 1$
- Variable K1 enables the add or subtract operation.
- If $X = 0$, then $\overline{X} = 1$ so $\overline{X}\,K1 = 1$, activating the addition of R1 and R2.
- If $X = 1$, then $X\,K1 = 1$, activating the addition of R1 and the two's complement of R2 (subtract).

Implementation of Add and Subtract Microoperations

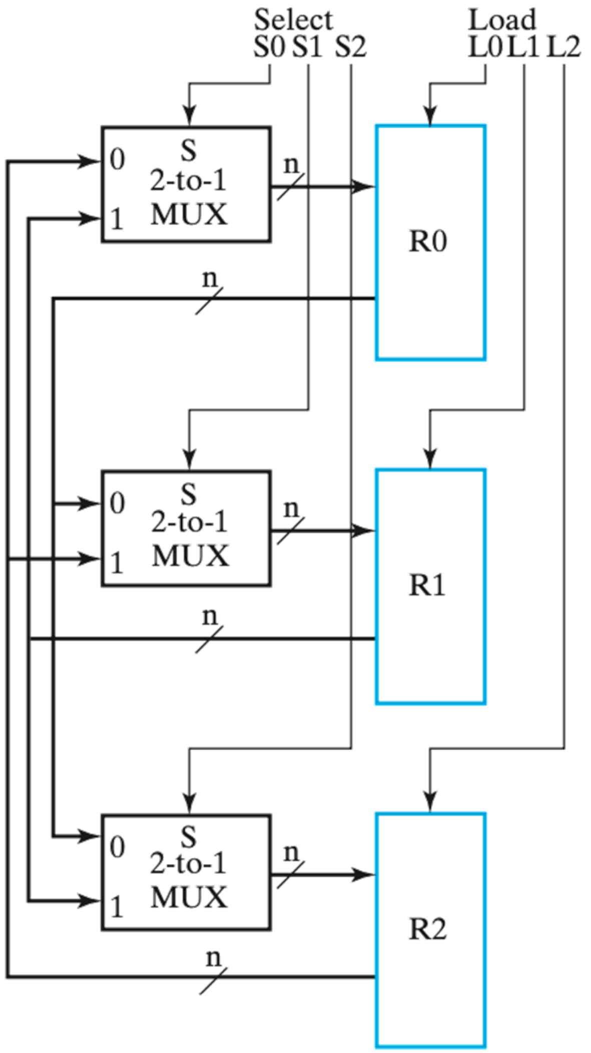
$\overline{X}\,K1: \; R1 \leftarrow R1 + R2$
$X\,K1: \; R1 \leftarrow R1 + \overline{R2} + 1$



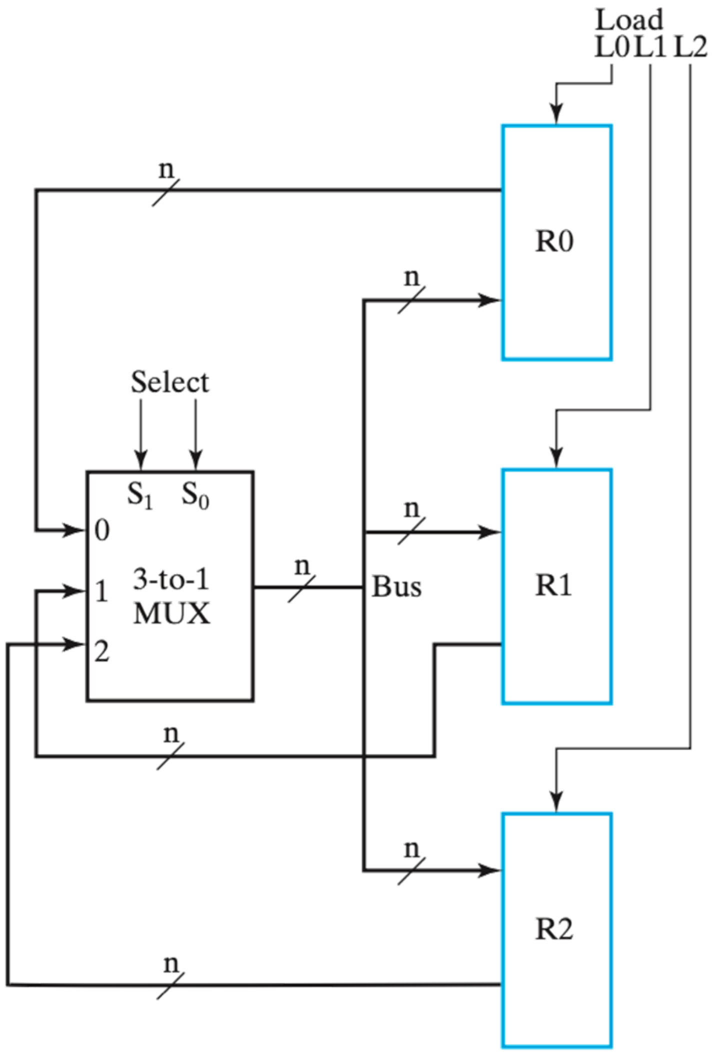
Register Transfer Structures

- **Multiplexer-Based Transfers** - Multiple inputs are selected by a multiplexer dedicated to the register, e.g.,
  - Shift registers
  - Counters
- Bus-Based Transfers - Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers
- Three-State Bus - Multiple inputs are selected by 3-state drivers with outputs connected to a bus that feeds multiple registers
- Other Transfer Structures - Use multiple multiplexers, multiple buses, and combinations of all the above

Dedicated Multiplexers vs. Single Bus

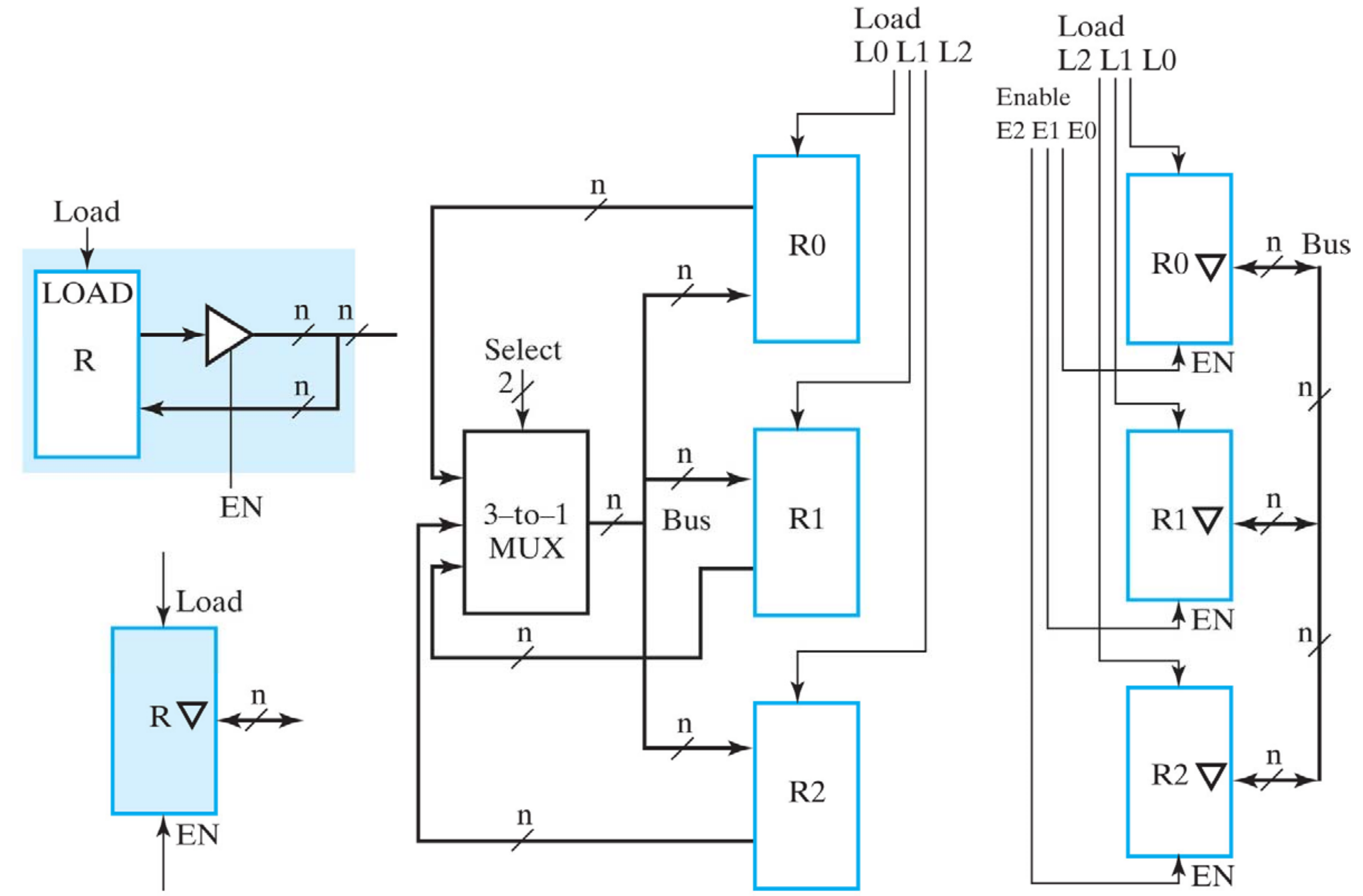


(a) Dedicated multiplexers



(b) Single bus

Multiplexer Bus vs. 3-State Bus



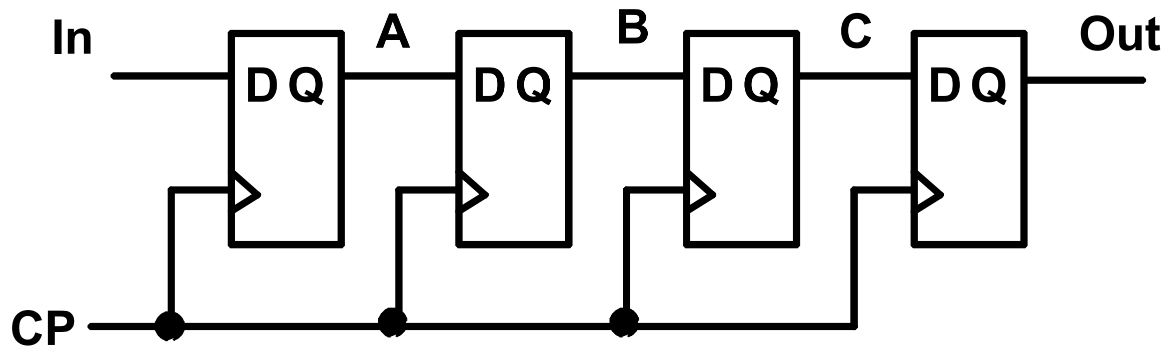
(a) Register with bidirectional input–output lines and symbol

(b) Multiplexer bus

(c) Three-state bus using registers with bidirectional lines

Shift Registers

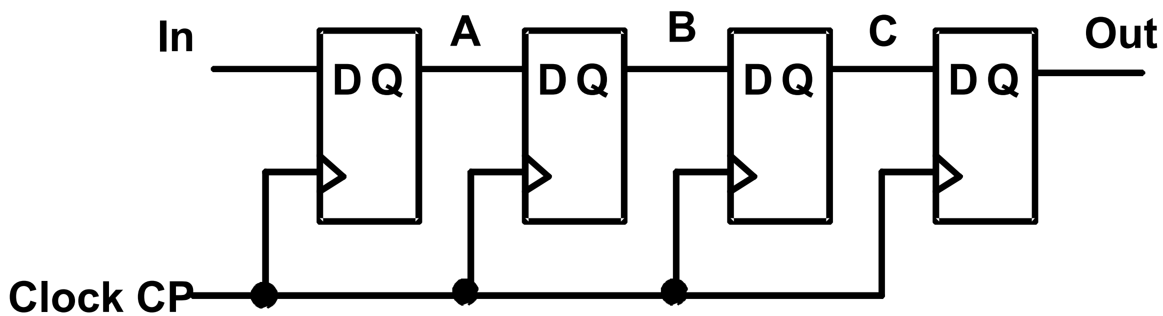
- A register capable of shifting its stored bits laterally in one or both directions is called a shift register
- Shift Registers move data laterally within the register toward its MSB or LSB position
- In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:



  - Data input, *In*, is called a serial input or the shift right input
  - Data output, *Out*, is often called the serial output
  - The vector (*A*, *B*, *C*, *Out*) is called the parallel output if they can be accessed in parallel

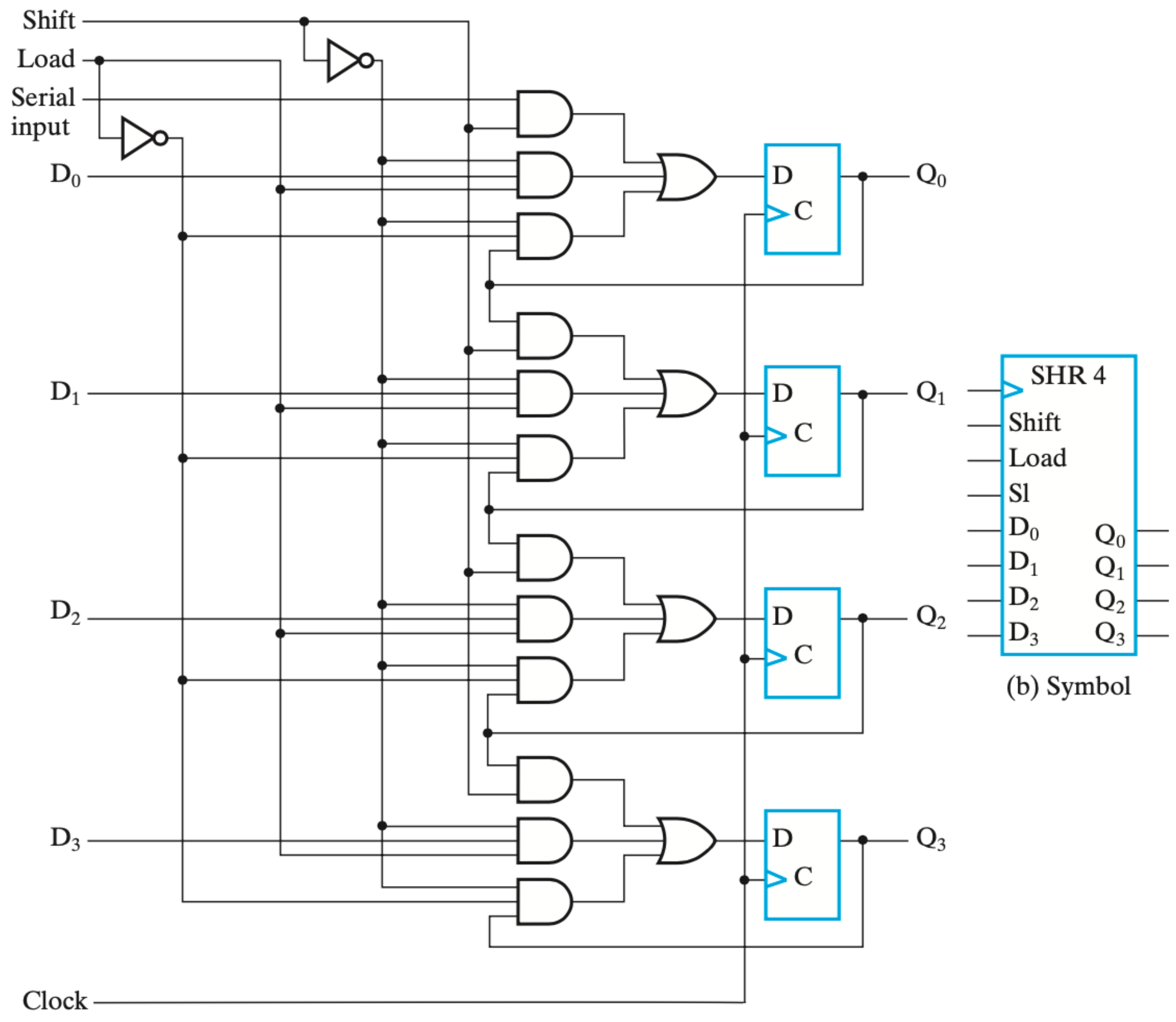
Shift Registers (cont'd)

- The behavior of the serial shift register is given in the listing on the lower right
- T0 is the register state just before the first clock pulse occurs
- T1 is after the first pulse and before the second.
- Initially unknown states are denoted by "?"
- Complete the last three rows of the table



CP	In	A	B	C	Out
T0	0	?	?	?	?
T1	1	0	?	?	?
T2	1	1	0	?	?
T3	0	1	1	0	?
T4	1				
T5	1				
T6	1				

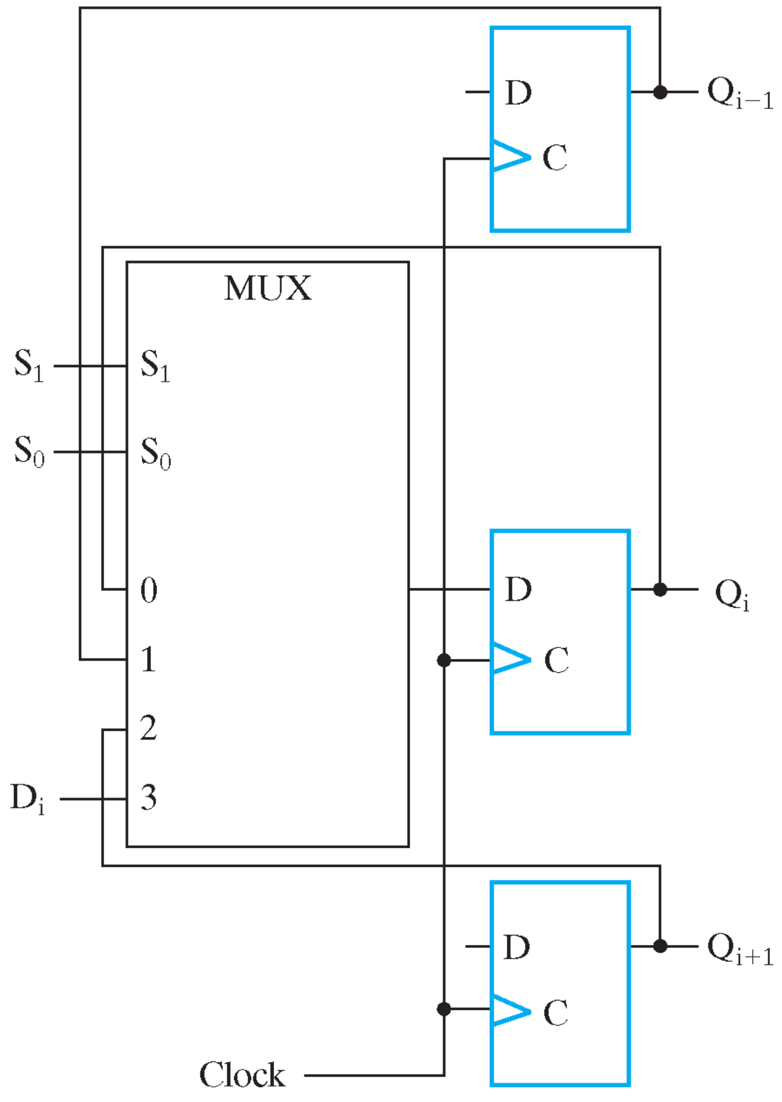
Shift Register with Parallel Load



(b) Symbol

Shift Registers with Additional Functions

- By placing a 4-input multiplexer in front of each D flip-flop in a shift register, we can implement a circuit with shifts right, shifts left, parallel load, hold.
- Shift registers can also be designed to shift more than a single bit position right or left
- Shift registers can be designed to shift a variable number of bit positions specified by a variable called a ***shift amount***.



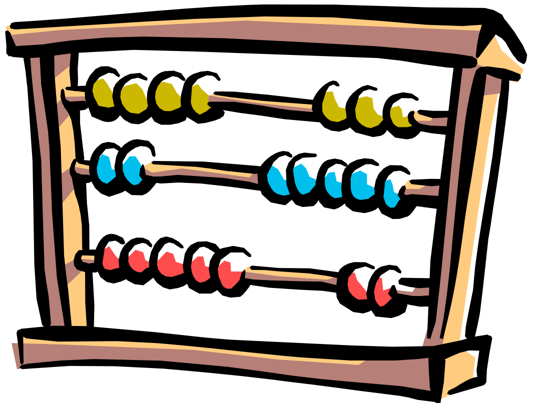
Classic Sequential Logic Elements

- Registers
  - Basic introduction
  - Registers in the digital system
  - Shift Registers

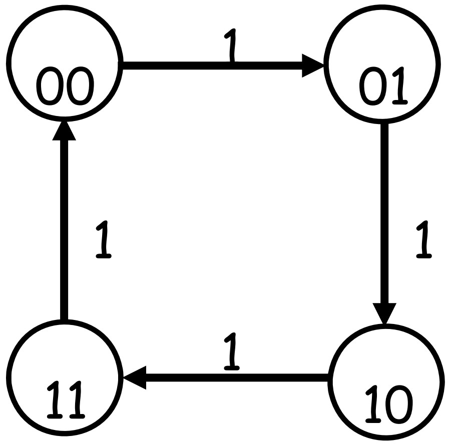
- Counters
  - Basic introduction
  - Ripple Counters
  - Synchronous Counters

Counters

- Counters are sequential circuits which "count" through a specific state sequence. They can count up, count down, or count through other fixed sequences.
- The state, or the flip-flop values themselves, serves as the "output."
- The output value increases by one on each clock cycle. After the largest value, the output "wraps around" back to 0.
- Using two bits, we'd get something like this:



Present State		Next State	
A	B	A	B
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



Counters (cont’d)

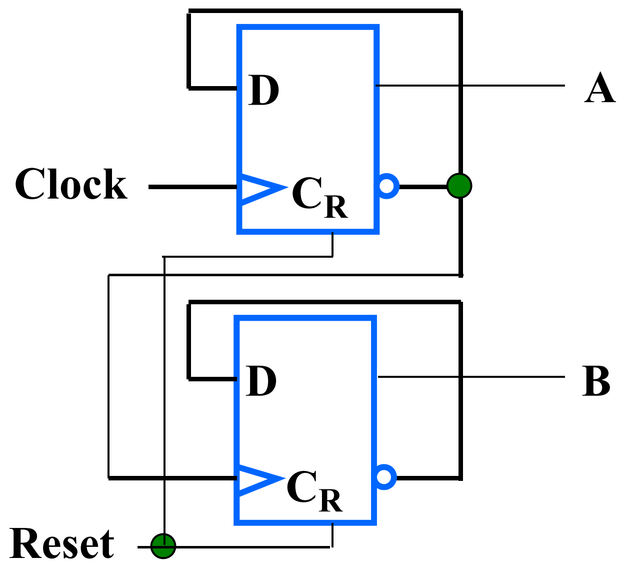
- Counters can act as simple clocks to keep track of “time.”
- You may need to record how many times something has happened.
  - How many bits have been sent or received?
  - How many steps have been performed in some computation?
- All processors contain a **program counter**, or **PC**.
  - Programs consist of a list of instructions that are to be executed one after another (for the most part).
  - The PC keeps track of the instruction currently being executed.
  - The PC increments once on each clock cycle, and the next program instruction is then executed.

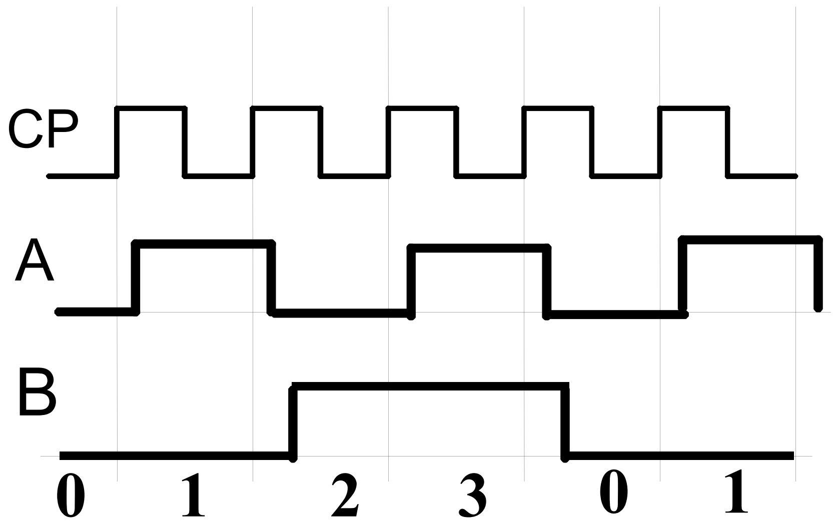
Counters (cont’d)

- Two distinct types are in common usage:
  - Ripple Counters
    - Clock connected to the flip-flop clock input on the LSB bit flip-flop
    - For all other bits, a flip-flop output is connected to the clock input, thus circuit is not truly synchronous!
    - Output change is delayed more for each bit toward the MSB.
    - Resurgent because of low power consumption
  - Synchronous Counters
    - Clock is directly connected to the flip-flop clock inputs
    - Logic is used to implement the desired state sequencing

Ripple Counters

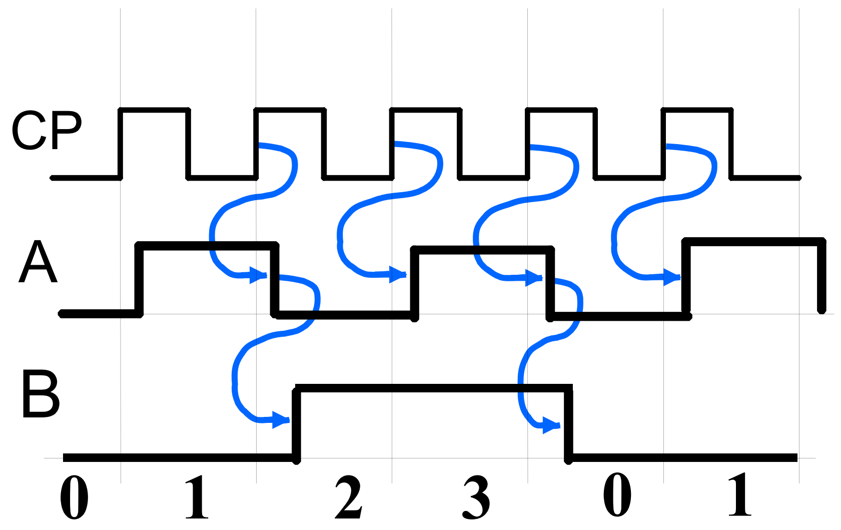
- When there is a positive edge on the clock input of A, A complements
- The clock input for flip-flop B is the complemented output of flip-flop A
- When flip A changes from 1 to 0, there is a positive edge on the clock input of B causing B to complement





Ripple Counters (cont'd)

- The arrows show the cause-effect relation-ship from the prior slide =>



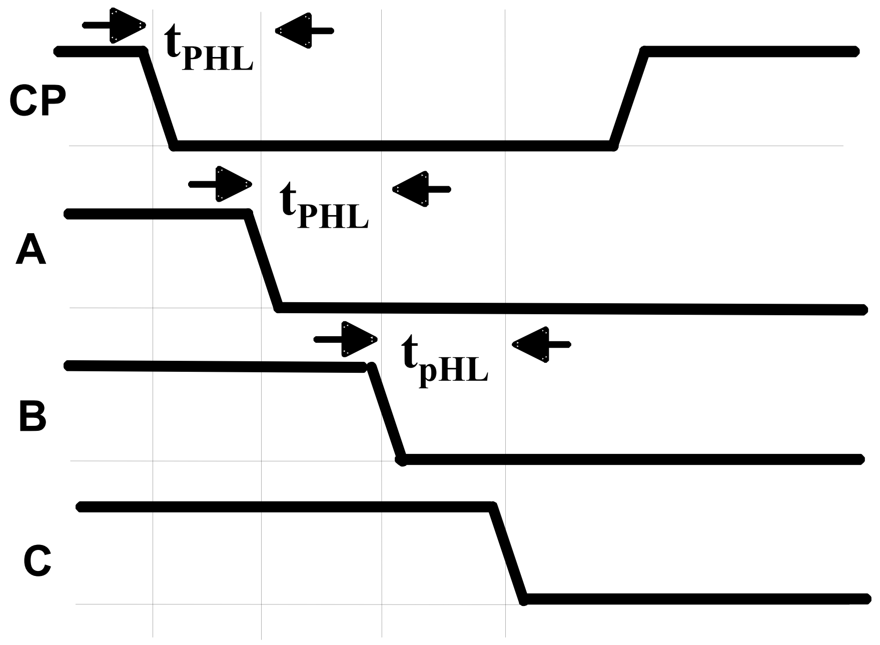
- The corresponding sequence of states => (B,A) = (0,0), (1,0), (1,1), (0,0), (0, 1), …
- Each additional bit, C, D, …behaves like bit B, changing half as frequently as the bit before it.
- For 3 bits: (C,B,A) = (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1), (0,0,0), …

Ripple Counters (cont’d)

- These circuits are called ripple counters because each edge sensitive transition (positive in the example) causes a change in the next flip-flop’s state.
- The changes “ripple” upward through the chain of flip-flops, i. e., each transition occurs after a clock-to-output delay from the stage before.
- To see this effect in detailed look at the waveforms on the next slide.

Ripple Counters (cont'd)

- Starting with C = B = A = 1, equivalent to (C,B,A) = 7 base 10, the next clock increments the count to (C,B,A) = 0 base 10. In fine timing detail:
  - The clock to output delay t_{PHL} causes an increasing delay from clock edge for each stage transition.
  - Thus, the count "ripples" from least to most significant bit.
  - For **n** bits, total worst case delay is $n \cdot t_{PHL}$

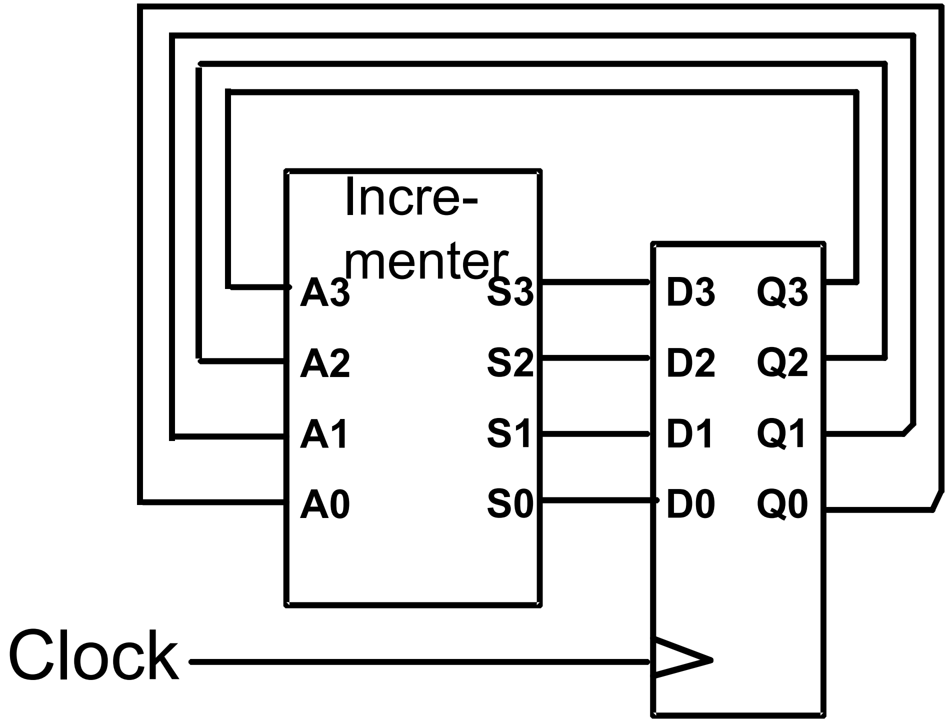


Synchronous Counters

- To eliminate the "ripple" effects, use a common clock for each flip-flop and a combinational circuit to generate the next state.
- For an up-counter, use an incrementer

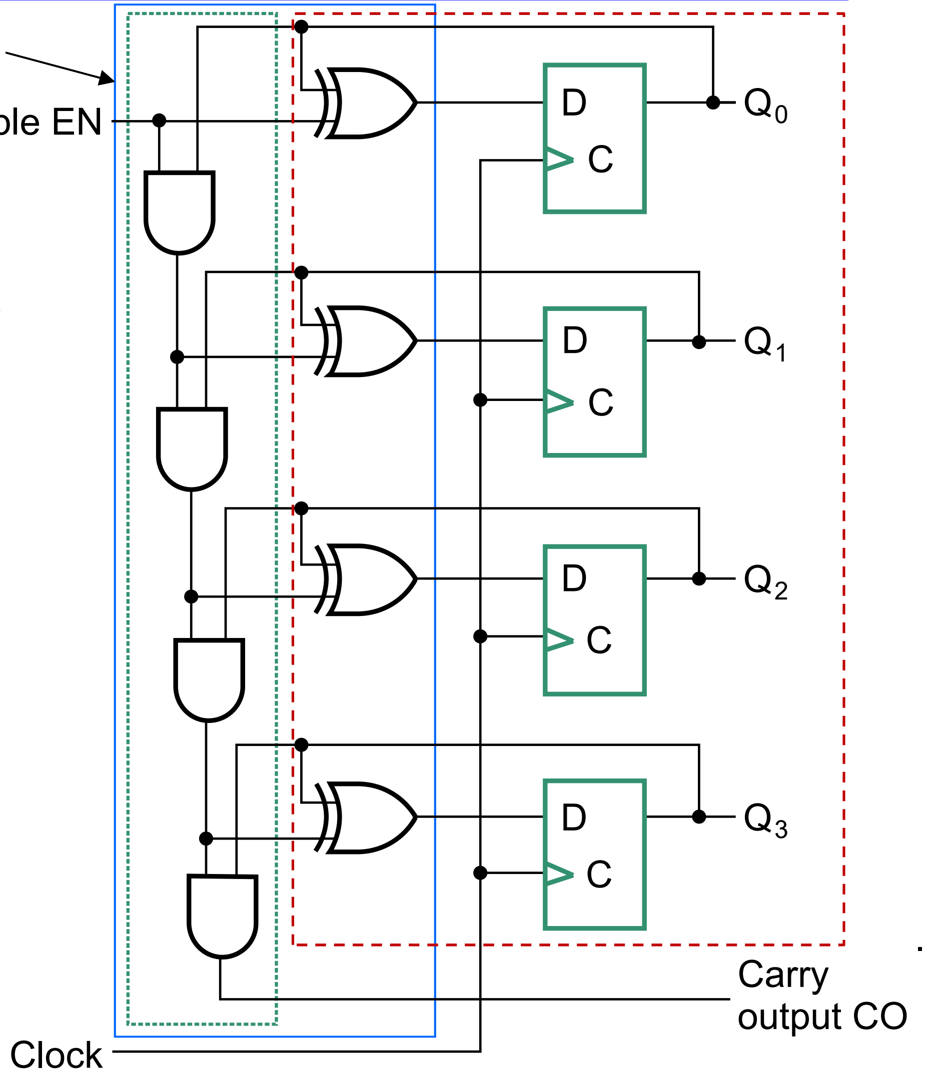
Upward Counting Sequence

Q_3	Q_2	Q_1	Q_0
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



Synchronous Counters (cont'd)

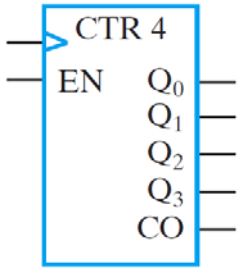
- Internal details => **Incrementer**
- Internal Logic
  - XOR complements each bit
  - AND chain causes complement of a bit if all bits toward LSB from it equal 1
- Count Enable
  - Forces all outputs of AND chain to 0 to "hold" the state
- Carry Out
  - Added as part of incrementer
  - Connect to Count Enable of additional 4-bit counters to form larger counters



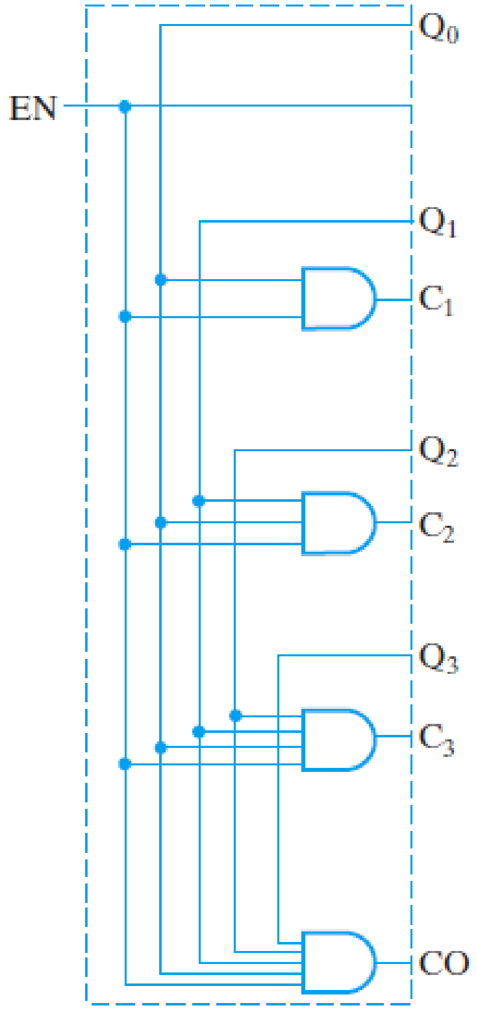
(a) Logic Diagram-Serial Gating

Synchronous Counters (cont'd)

- Carry chain
  - Series of AND gates through which the carry "ripples"
  - Yields long path delays
  - Called serial gating
- Replace AND carry chain with ANDs => in parallel
  - Reduces path delays
  - Called parallel gating
  - Like carry lookahead
  - Lookahead can be used on COs and ENs to prevent long paths in large counters
- Symbol for synchronous counter



(c) Symbol



(b) Logic diagram—parallel gating

Other Counters

- Divide-by-n (Modulo n) Counter
  - Count is remainder of division by n; n may not be a power of 2 or
  - Count is arbitrary sequence of n states specifically designed state-by-state
  - Includes modulo 10 which is the BCD counter

Synchronous BCD with D Flip-Flops

- Use the sequential logic model to design a synchronous BCD counter with D flip-flops
- Input combinations 1010 through 1111 are don't cares

Present State				Next State				Output
Q_8	Q_4	Q_2	Q_1	$D_8 = Q_8(t+1)$	$D_4 = Q_4(t+1)$	$D_2 = Q_2(t+1)$	$D_1 = Q_1(t+1)$	Y
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	1	0
0	0	1	1	0	1	0	0	0
0	1	0	0	0	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	0	1	0	0	0	0	1

Synchronous BCD with D Flip-Flops (cont'd)

- Use K-Maps to two-level optimize the next state equations and manipulate them on necessary:

$$\begin{aligned} D_1 &= \overline{Q_1} \\ D_2 &= \overline{Q_8}\,\overline{Q_2}Q_1 + Q_2\overline{Q_1} \\ D_4 &= \overline{Q_4}Q_2Q_1 + Q_4\overline{Q_2} + Q_4\overline{Q_1} \\ D_8 &= Q_8\overline{Q_1} + Q_4Q_2Q_1 \end{aligned}$$

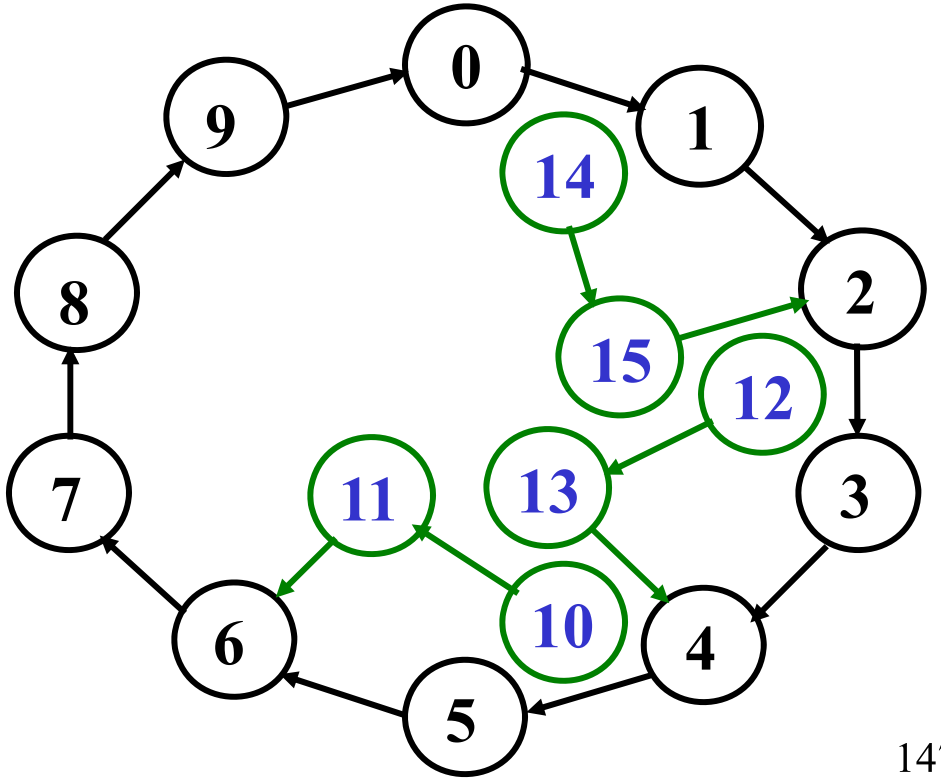
$$Y = ?$$

- The logic diagram can be draw from these equations
  - An asynchronous or synchronous reset should be added
- What happens if the counter is perturbed by a power disturbance or other interference, and it enters a state other than 0000 through 1001?

Synchronous BCD with D Flip-Flops (cont'd)

- Find the actual values of the six next states for the don't care combinations from the equations
- Find the overall state diagram to assess behavior for the don't care states (states in decimal)

Present State				Next State			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1
1	0	1	0	1	0	1	1
1	0	1	1	0	1	1	0
1	1	0	0	1	1	0	1
1	1	0	1	0	1	0	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	1	0



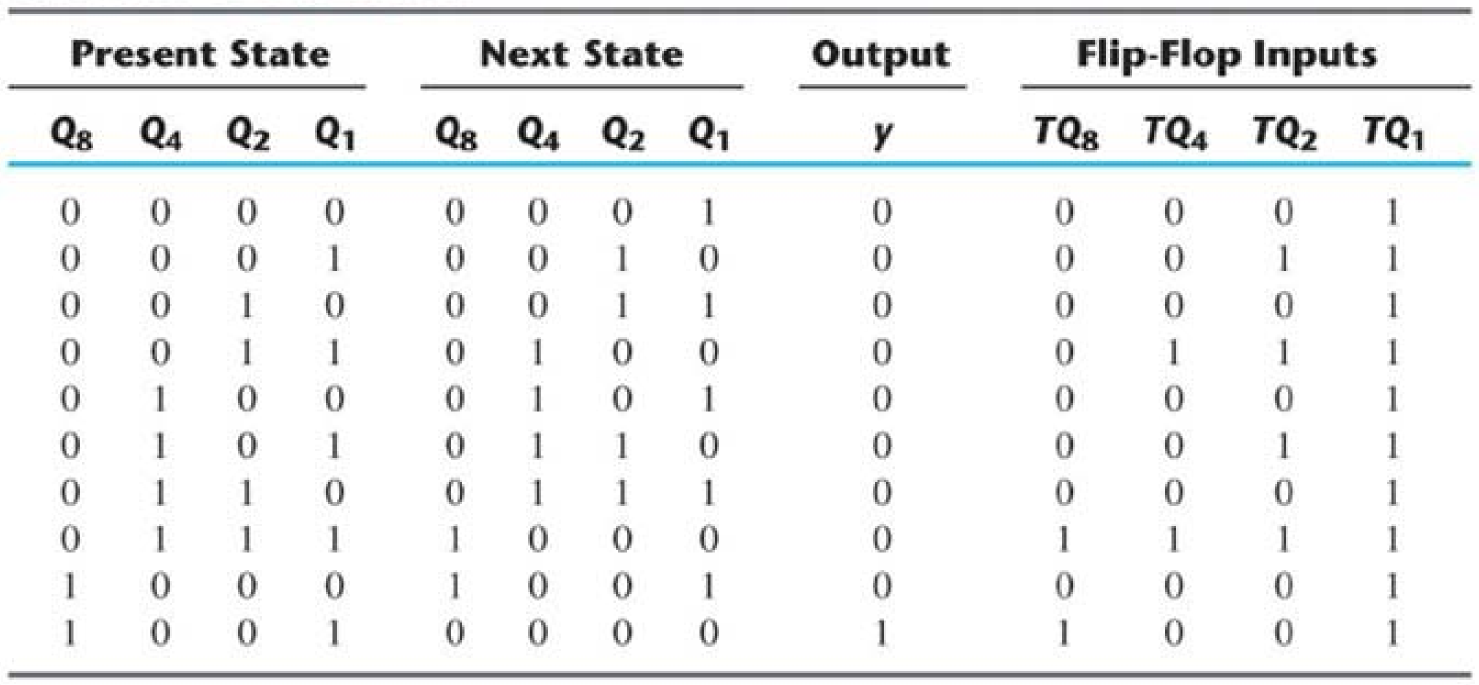
Synchronous BCD with D Flip-Flops (cont’d)

- For the BCD counter design, if an invalid state is entered, return to a valid state occurs within two clock cycles
- Is this adequate? If not:
  - Is a signal needed that indicates that an invalid state has been entered? What is the equation for such a signal?
  - Does the design need to be modified to return from an invalid state to a valid state in one clock cycle?
  - Does the design need to be modified to return from an invalid state to a specific state (such as 0)?
- The action to be taken depends on:
  - The application of the circuit
  - Design group policy

Synchronous BCD with T Flip-Flops

- Use the sequential logic model to design a synchronous BCD counter with T flip-flops
- Input combinations 1010 through 1111 are don’t cares

State Table for BCD Counter



Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

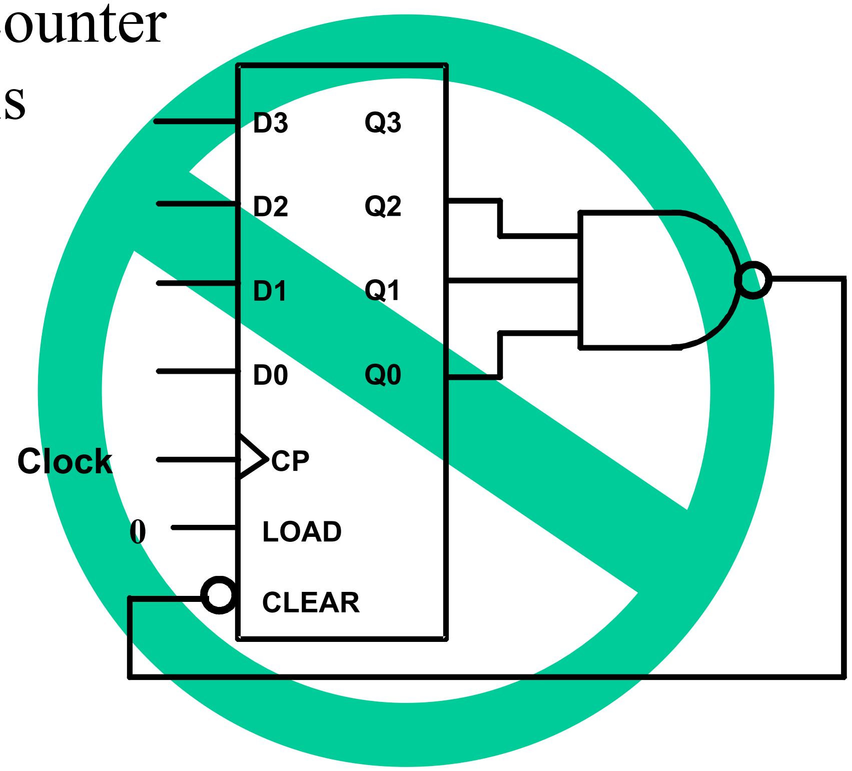
What about the equations?

Counting Modulo N

- The following techniques use an *n*-bit binary counter with asynchronous or synchronous clear and/or parallel load:
  - Detect a *terminal count* of N in a Modulo-N count sequence to asynchronously Clear the count to 0 or asynchronously Load in value 0 (These lead to counts which are present for only a very short time and can fail to work for some timing conditions!)
  - Detect a terminal count of N - 1 in a Modulo-N count sequence to Clear the count synchronously to 0
  - Detect a terminal count of N - 1 in a Modulo-N count sequence to synchronously Load in value 0
  - Detect a terminal count and use Load to preset a count of the terminal count value minus (N - 1)
- Alternatively, custom design a modulo N counter as done for BCD

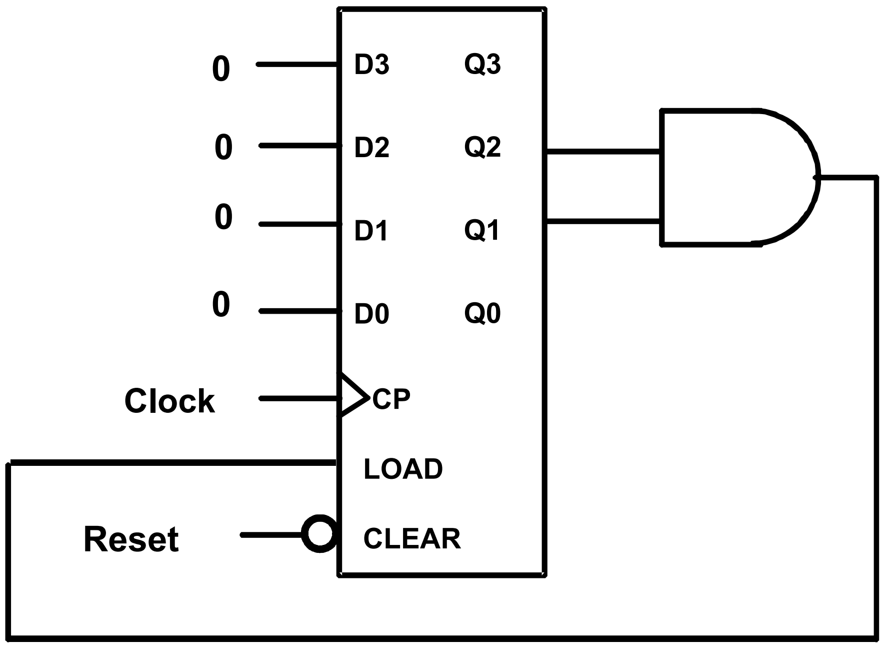
Counting Modulo 7: Detect 7 and Asynchronously Clear

- A synchronous 4-bit binary counter with an asynchronous Clear is used to make a Modulo 7 counter.
- Use the Clear feature to detect the count 7 and clear the count to 0. This gives a count of 0, 1, 2, 3, 4, 5, 6, 7(short)0, 1, 2, 3, 4, 5, 6, 7(short)0, etc.
- DON'T DO THIS! Referred to as a "suicide" counter! (Count "7" is "killed," but the designer's job may be dead as well!)



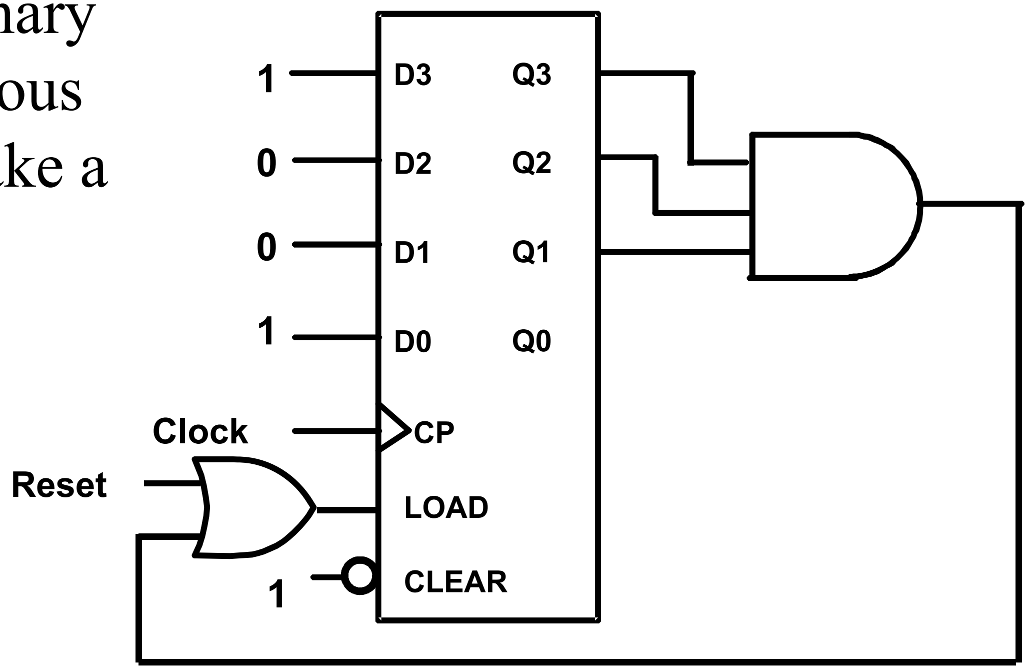
Counting Modulo 7: Synchronously Load on Terminal Count of 6

- A synchronous 4-bit binary counter with a synchronous load and an asynchronous clear is used to make a Modulo 7 counter
- Use the Load feature to detect the count "6" and load in "zero". This gives a count of 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, ...
- Using don't cares for states above 0110, detection of 6 can be done with Load = Q2 Q1



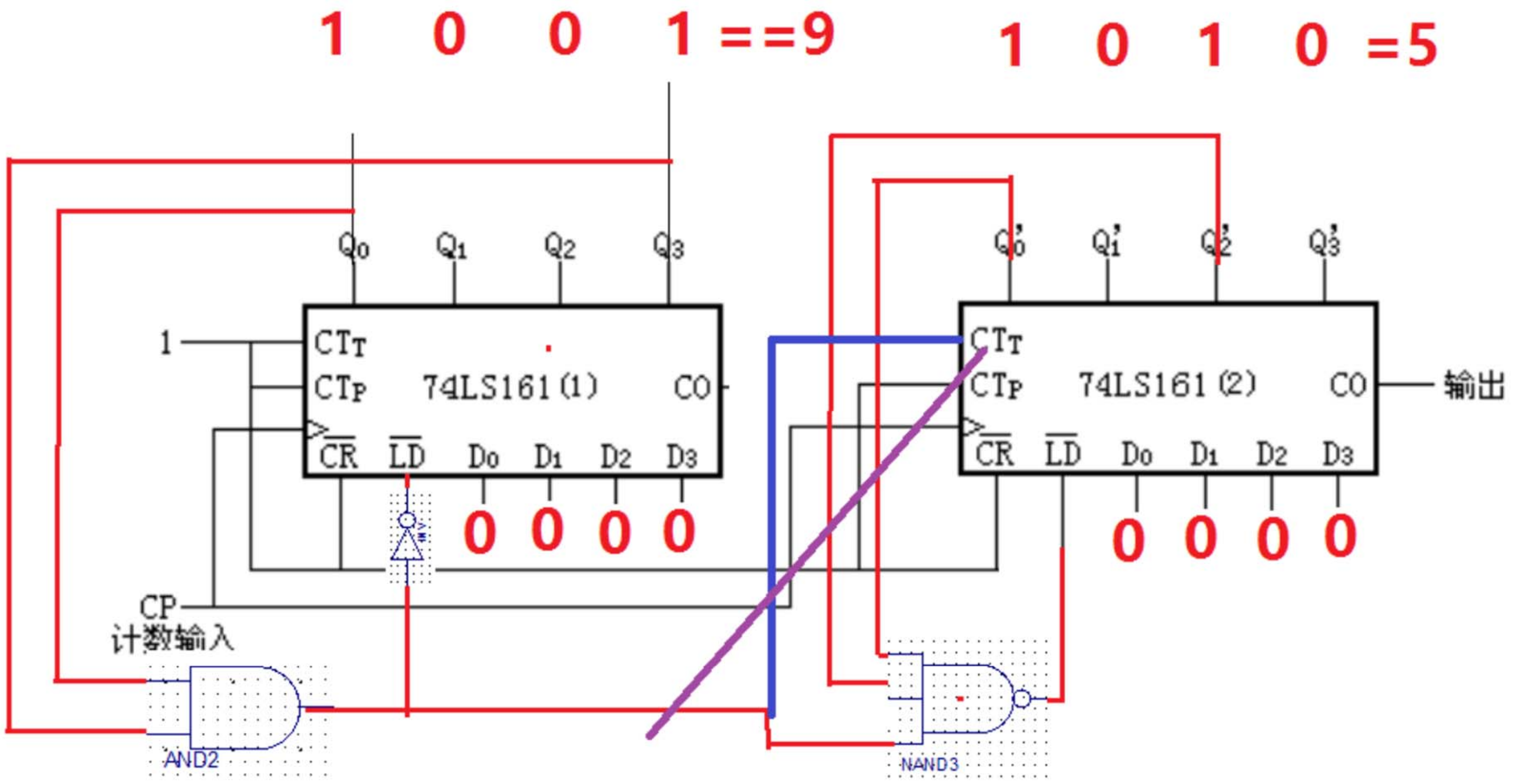
Counting Modulo 6: Synchronously Preset 9 on Reset and Load 9 on Terminal Count 14

- A synchronous, 4-bit binary counter with a synchronous Load is to be used to make a Modulo 6 counter.
- Use the Load feature to preset the count to 9 on Reset and detection of count 14.



- This gives a count of 9, 10, 11, 12, 13, 14, 9, 10, 11, 12, 13, 14, 9, …
- If the terminal count is 15 detection is usually built in as Carry Out (CO)

分钟60进制（十进制显示）



CTT=1 displaynumber[7:4]+1

分钟60进制说明

END