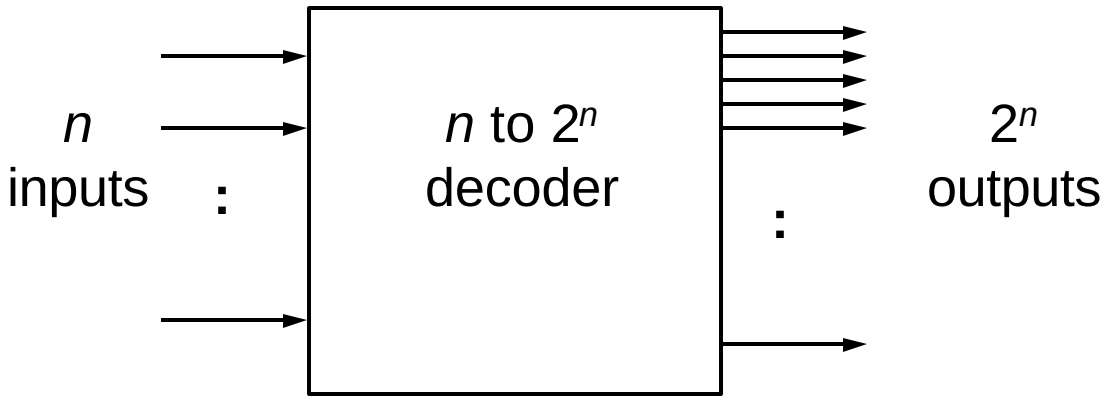
Final Exam Review

- **Combinational Logic Building Blocks:**
  - **Decoders, Encoders, Multiplexers, Demultiplexers**
  - ***Implementing functions using decoders, multiplexers.***
- **Combinational Arithmetic Circuits:**
  - **Adders, Subtractors, Multipliers, Comparators, shifters.**
- **Sequential Logic Circuits:**
  - **Latches, Flip-Flips.**
- **Clocked Synchronous State Machines:**
  - ***State Machine Analysis***
  - ***State Machine Design***
- **Registers & Counters.**

Binary n-to-2^n Decoders

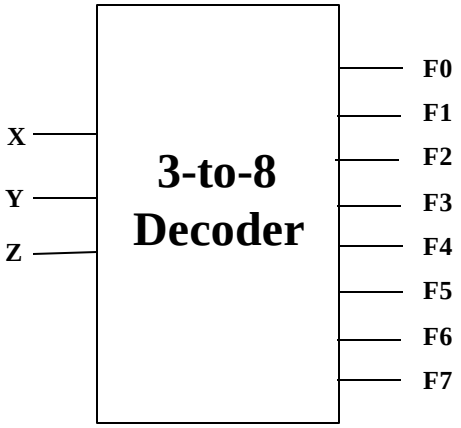
- **A binary decoder has n inputs and 2^n outputs.**
- **Only the output corresponding to the input value is equal to 1.**

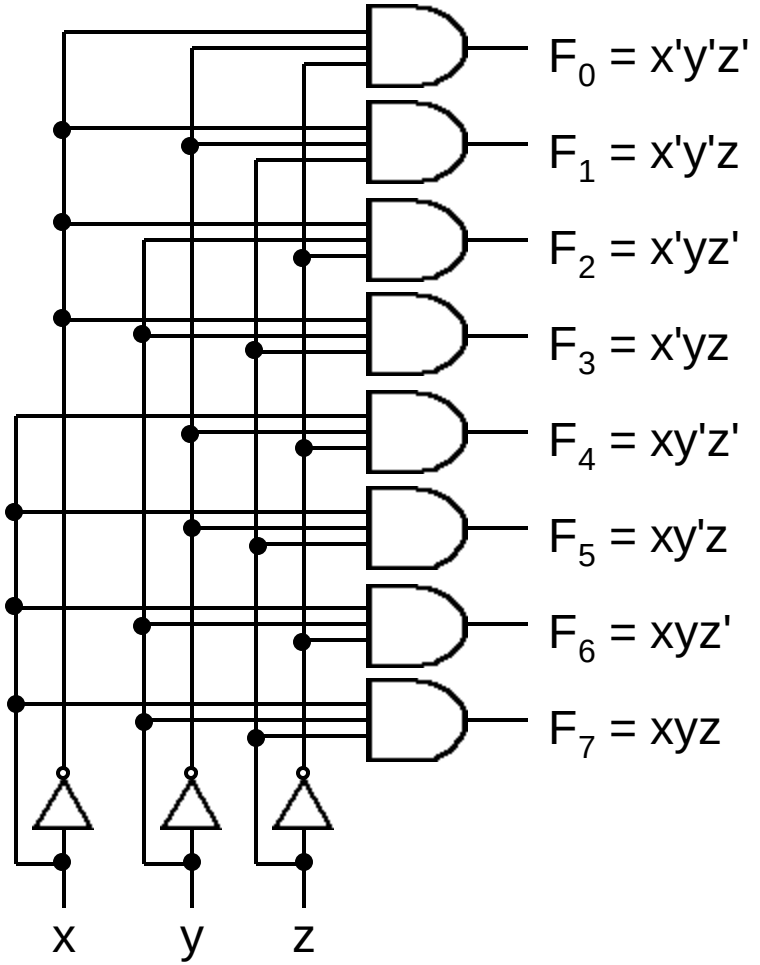


3-to-8 Binary Decoder

Truth Table:

x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1





Implementing Functions Using Decoders

- **Any n-variable logic function, in canonical sum-of-minterms form can be implemented using a single n-to-2^n decoder to generate the minterms, and an OR gate to form the sum.**
  - The output lines of the decoder corresponding to the minterms of the function are used as inputs to the or gate.
- **Any combinational circuit with *n* inputs and *m* outputs can be implemented with an n-to-2^n decoder with *m* OR gates.**
- **Suitable when a circuit has many outputs, and each output function is expressed with few minterms.**

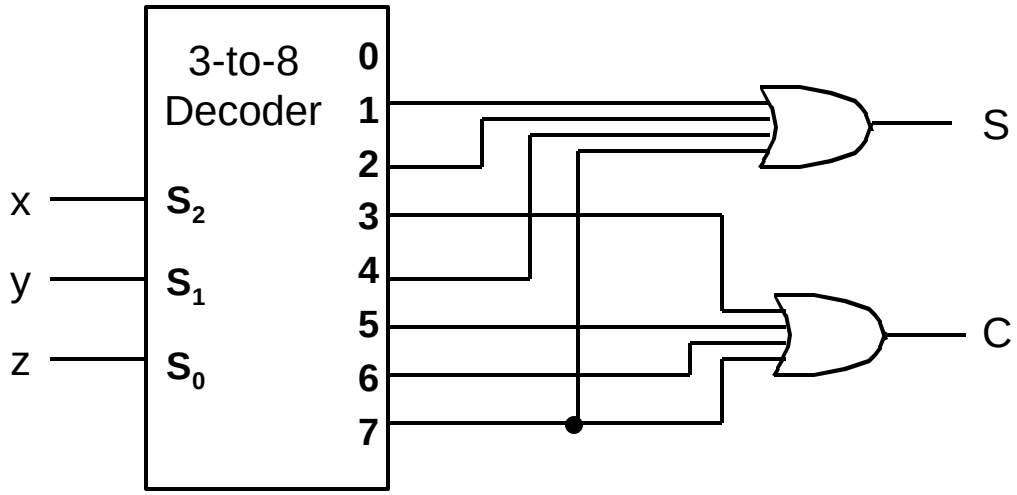
Implementing Functions Using Decoders

- **Example: Full adder**

$S(x, y, z) = \Sigma (1,2,4,7)$

$C(x, y, z) = \Sigma (3,5,6,7)$

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Encoders

- If the a decoder's output code has fewer bits than the input code, the device is usually called an encoder.

  e.g. 2^n-to-n, priority encoders.

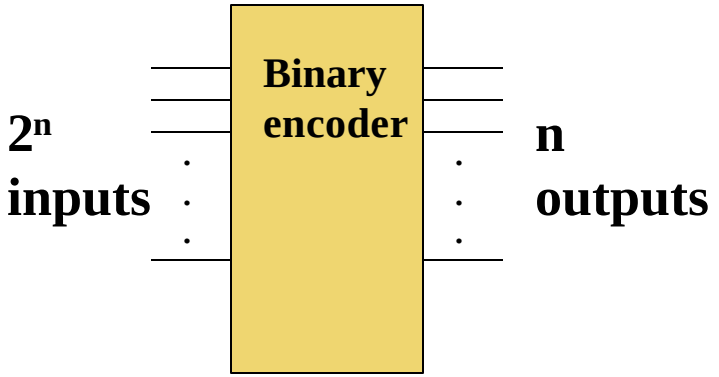
- The simplest encoder is a 2^n-to-n binary encoder, where it has only one of 2^n inputs = 1 and the output is the n-bit binary number corresponding to the active input.

- For an 8-to-3 binay encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

$$Y0 = I1 + I3 + I5 + I7$$

$$Y1 = I2 + I3 + I6 + I7$$

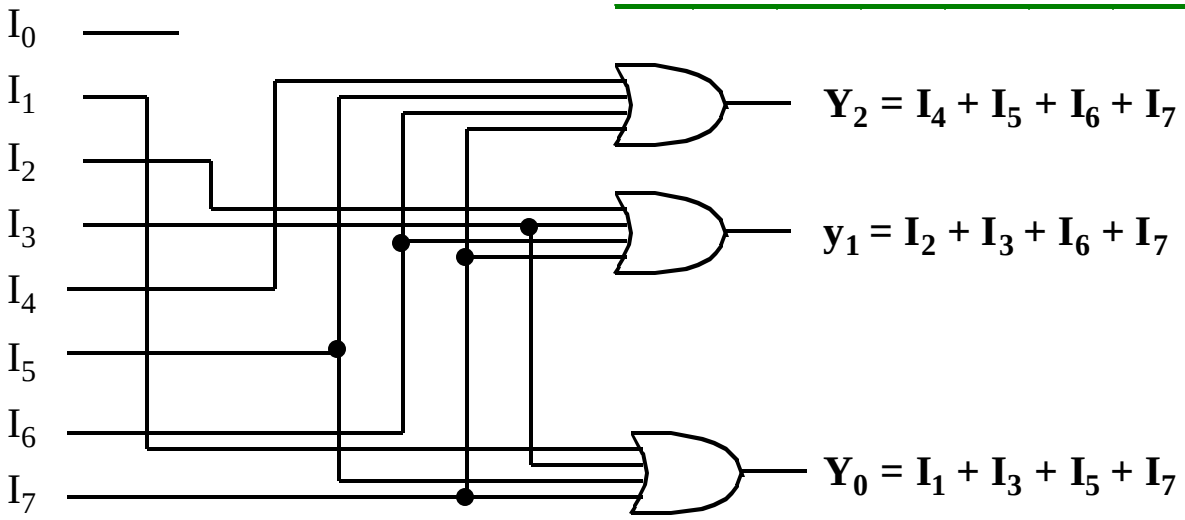
$$Y2 = I4 + I5 + I6 + I7$$



8-to-3 Binary Encoder

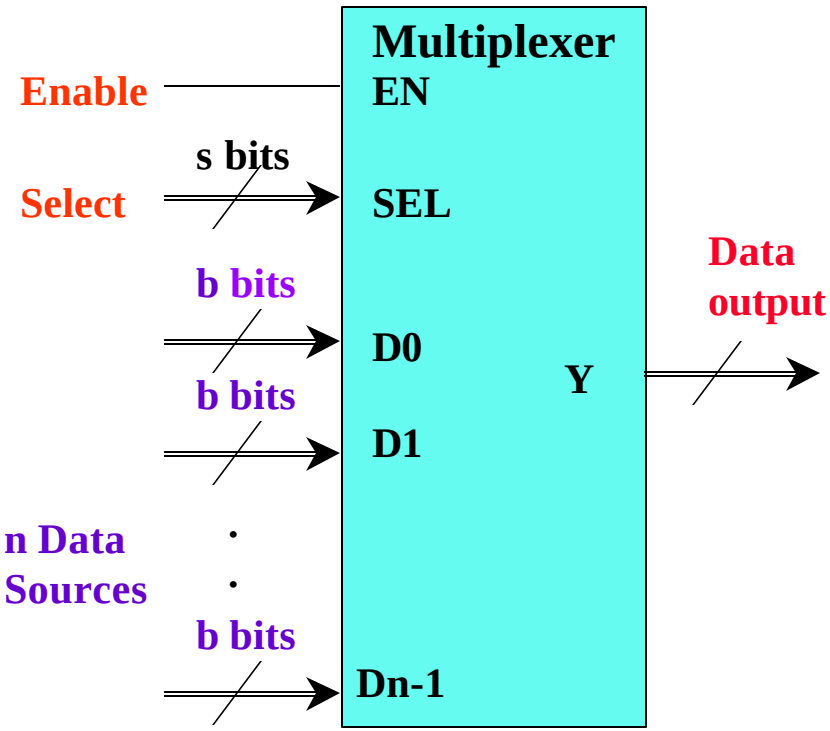
At any one time, only one input line has a value of 1.

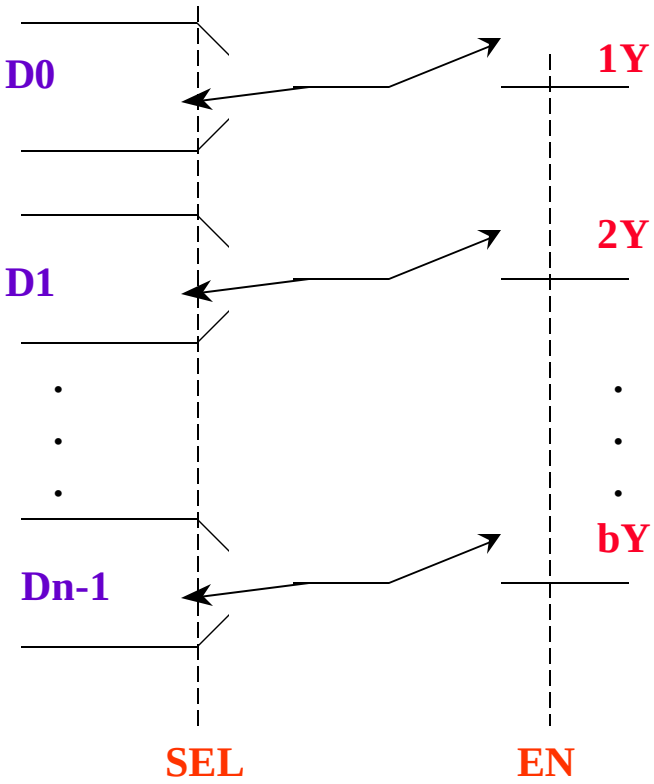
Inputs								Outputs		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	y_2	y_1	y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1



Multiplexers

- A multiplexer (MUX) is a digital switches which connects data from one of n sources to the output.
- A number of select inputs determine which data source is connected to the output.



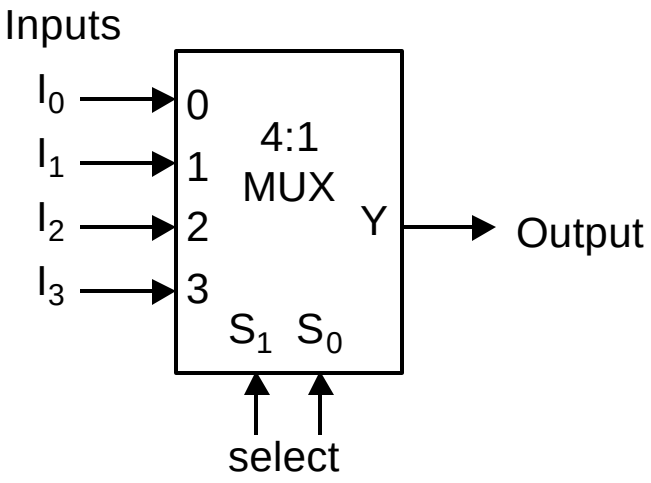


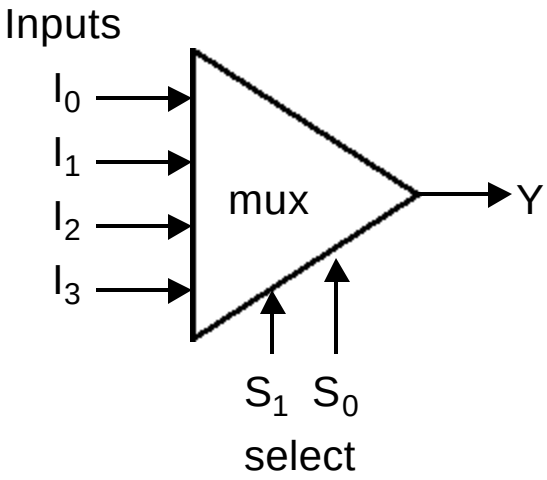
4-to-1 MUX

Truth table for a 4-to-1 multiplexer:

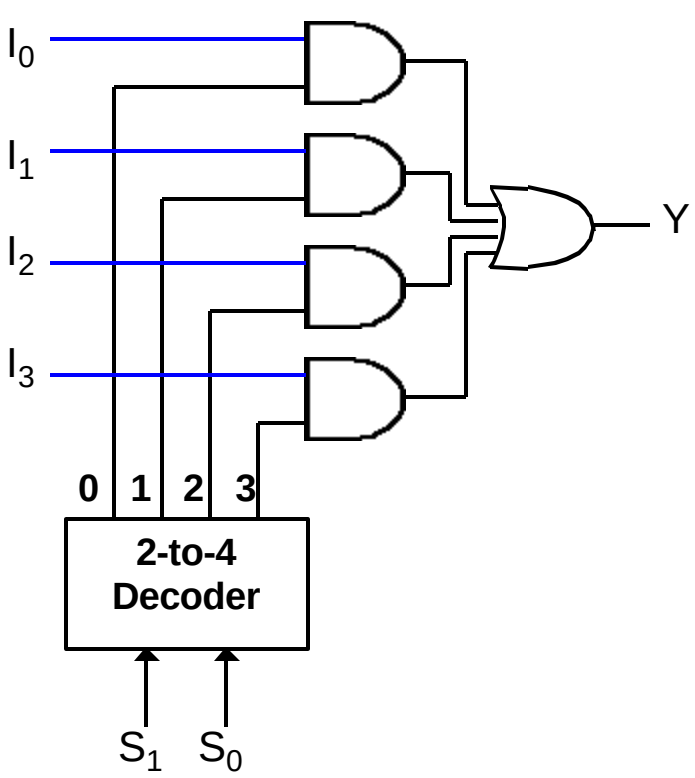
I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

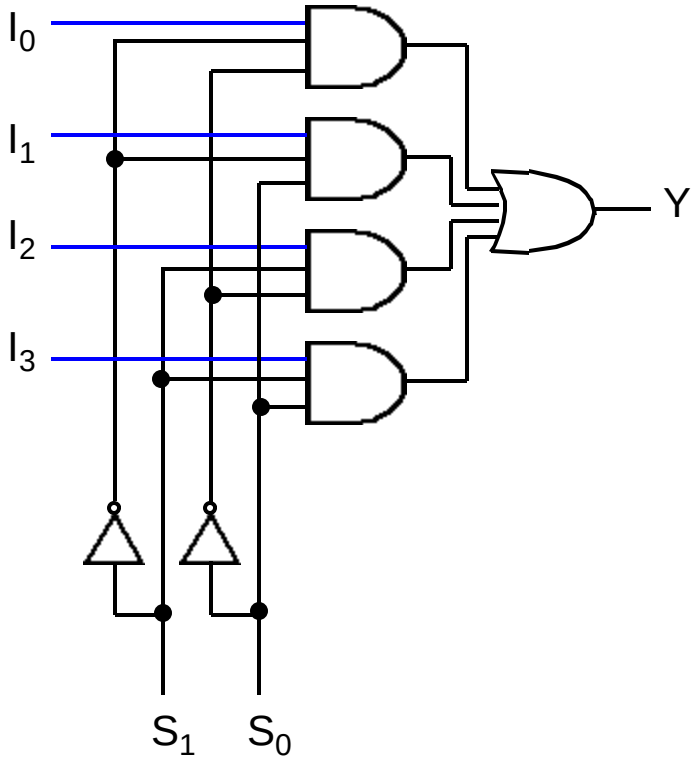
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3





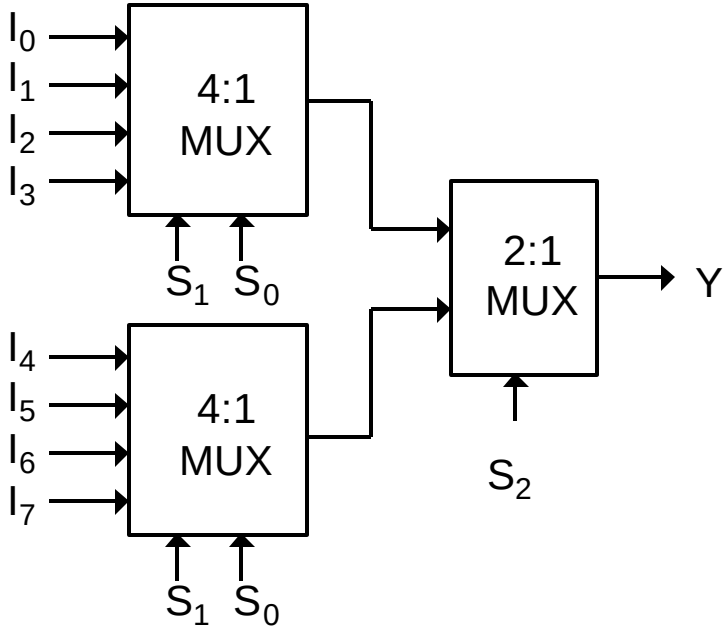
4-to-1 MUX Circuit





Larger Multiplexers

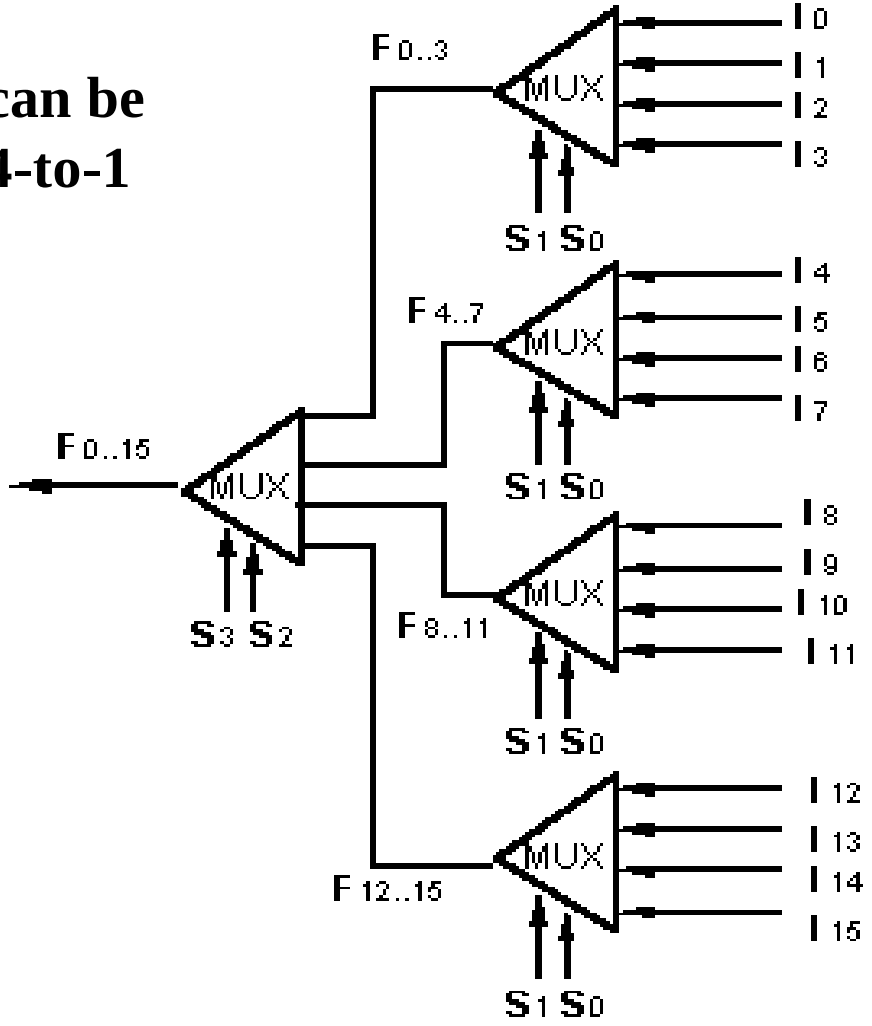
- **Larger multiplexers can be constructed from smaller ones.**
- **An 8-to-1 multiplexer can be constructed from smaller multiplexers as shown:**



S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

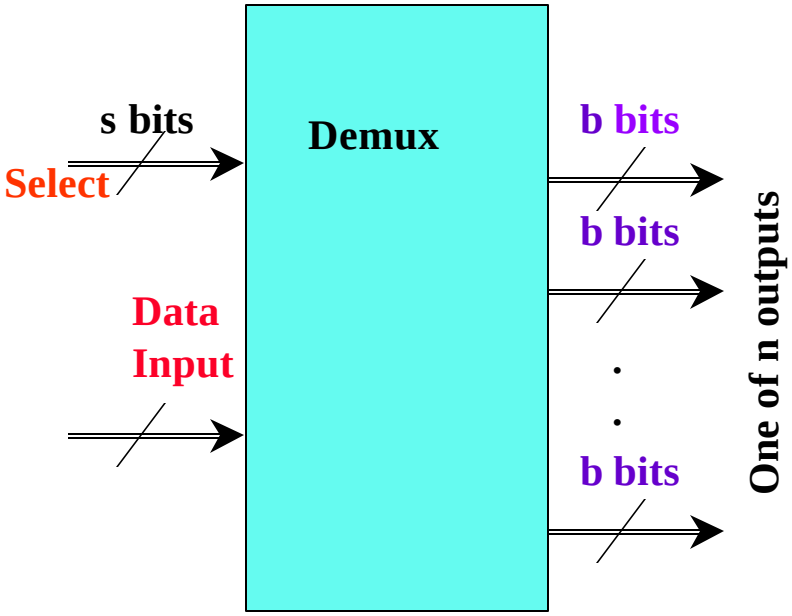
Larger Multiplexers

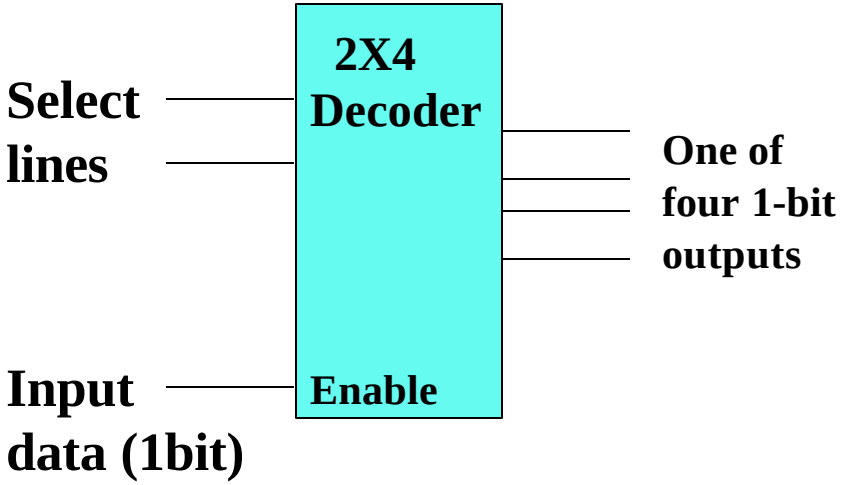
- **A 16-to-1 multiplexer can be constructed from five 4-to-1 multiplexers:**



Demultiplexers

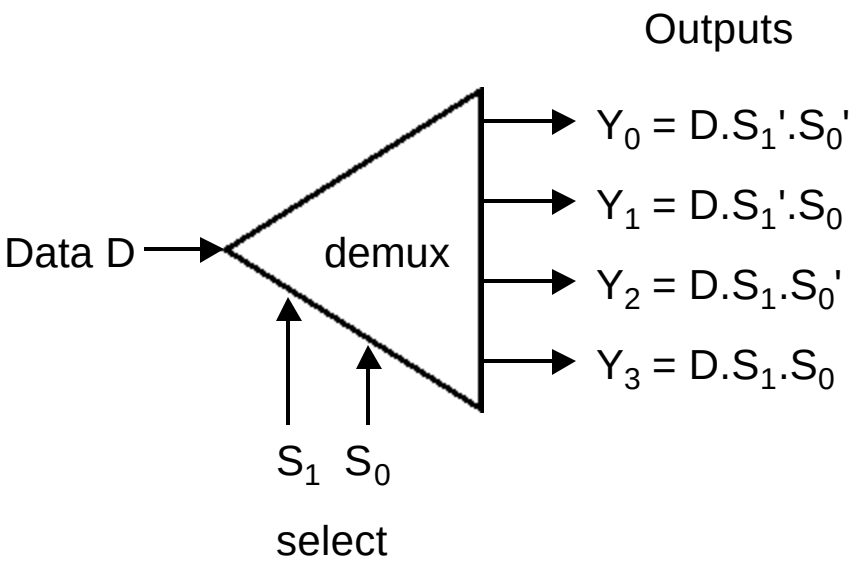
- Digital switches to connect data from one input source to one of n outputs.
- Usually implemented by using n-to-2^n binary decoders where the decoder's enable line is used for data input of the demultiplexer.



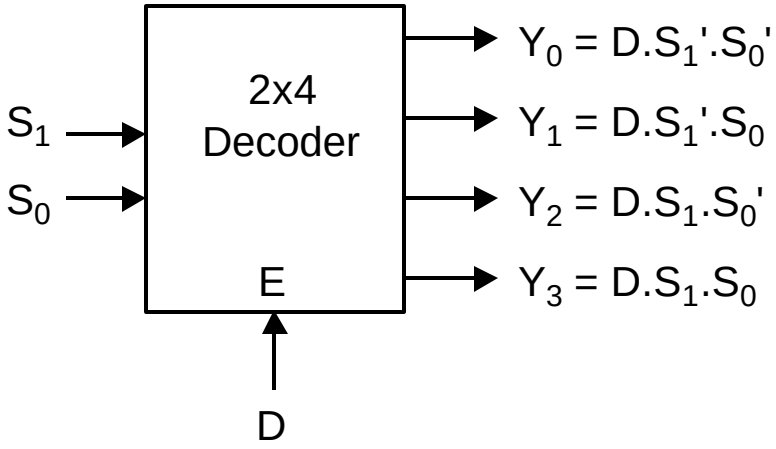


1-bit 4-output demultiplexer using a 2x4 binary decoder.

1-to-4 Demultiplexer



S_1	S_o	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

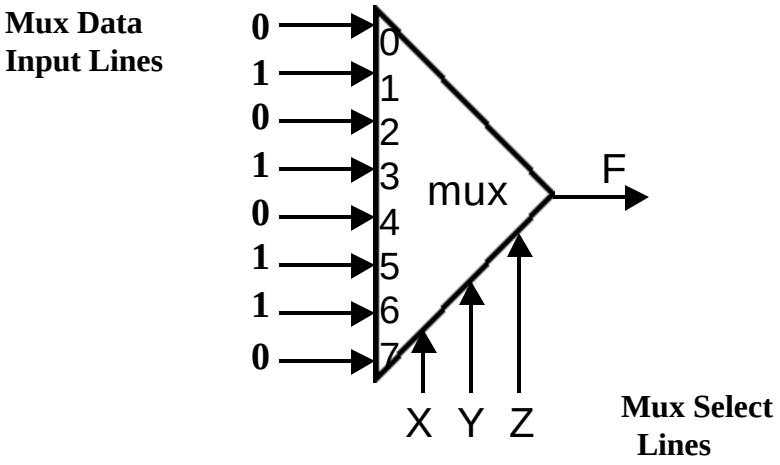


Implementing n-variable Functions Using 2^n-to-1 Multiplexers

- **Any n-variable logic function, in canonical sum-of-minterms form can be implemented using a single 2^n-to-1 multiplexer:**
  - **The n input variables are connected to the mux select lines.**
  - **For each mux data input line I_i ($0 \leq i \leq 2^n - 1$):**
    - **Connect 1 to mux input line I_i if i is a minterm of the function.**
    - **Otherwise, connect 0 to mux input line I_i (because i is not a minterm of the function thus the selected input should be 0).**

Example: 3-variable Function Using 8-to-1 mux

- **Implement the function F(X,Y,Z) = S(1,3,5,6) using an 8-to-1 mux.**
  - **Connect the input variables X, Y, Z to mux select lines.**
  - **Mux data input lines 1, 3, 5, 6 that correspond to function minterms are connected to 1.**
  - **The remaining mux data input lines 0, 2, 4, 7 are connected to 0.**



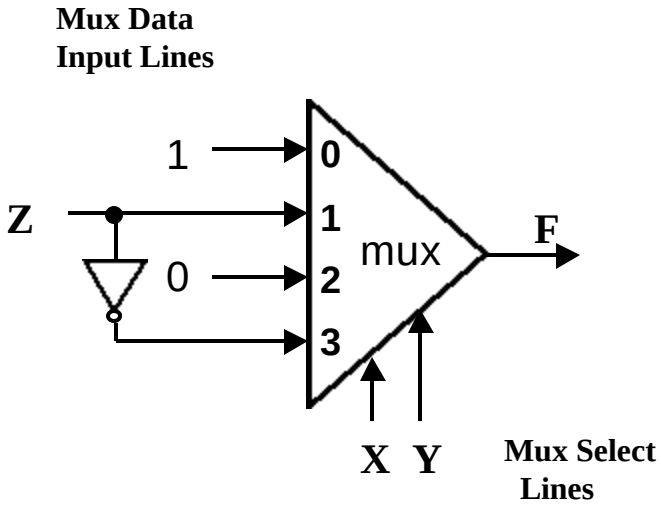
Implementing n-variable Functions Using 2^{n-1}-to-1 Multiplexers

- Any n-variable logic function can be implemented using a smaller 2^{n-1}-to-1 multiplexer and a single inverter (e.g 4-to-1 mux to implement 3 variable functions) as follows:
  - Express function in canonical sum-of-minterms form.
  - Choose n-1 variables as inputs to mux select lines.
  - Construct the truth table for the function, but grouping inputs by selection line values (i.e select lines as most significant inputs).
  - Determine multiplexer input line i values by comparing the remaining input variable and the function F for the corresponding selection lines value i:
    - Four possible mux input line i values:
      - Connect to 0 if the function is 0 for both values of remaining variable.
      - Connect to 1 if the function is 1 for both values of remaining variable.
      - Connect to remaining variable if function is equal to the remaining variable.
      - Connect to the inverted remaining variable if the function is equal to the remaining variable inverted.

Example: 3-variable Function Using 4-to-1 mux

- **Implement the function F(X,Y,Z) = S(0,1,3,6) using a single 4-to-1 mux and an inverter.**
  - **We choose the two most significant inputs X, Y as mux select lines.**
  - **Construct truth table:**

Select Lines Value i	Select Lines X	Y	Z	F	Mux Input i
0	0	0	0	1	1
	0	0	1	1	
1	0	1	0	0	Z
	0	1	1	1	
2	1	0	0	0	0
	1	0	1	0	
3	1	1	0	1	Z’
	1	1	1	0	



- **We Determine multiplexer input line i values by comparing the remaining input variable Z and the function F for the corresponding selection lines value i:**
  - **when XY=00 the function F=1 (for both Z=0, Z=1) thus mux input0 = 1**
  - **when XY=01 the function F=Z thus mux input1 = Z**
  - **when XY=10 the function F=0 (for both Z=0, Z=1) thus mux input2 = 0**
  - **when XY=11 the function F=Z’ thus mux input3 = Z’**

Combinational Arithmetic Circuits

- **Addition:**
  - **Half Adder (HA).**
  - **Full Adder (FA).**
  - **Carry Ripple Adders.**
  - **Carry Look-Ahead Adders.**
- **Subtraction:**
  - **Half Subtractor.**
  - **Full Subtractor.**
  - **Borrow Ripple Subtractors.**
  - **Subtraction using adders.**
- **Multiplication:**
  - **Combinational Array Multipliers.**

Full Adder

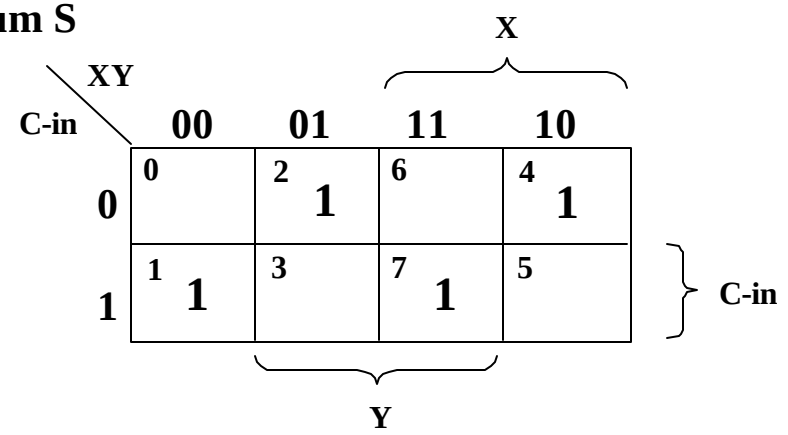
- **Adding two single-bit binary values, X, Y with a carry input bit C-in produces a sum bit S and a carry out C-out bit.**

Full Adder Truth Table

Inputs			Outputs	
X	Y	C-in	S	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

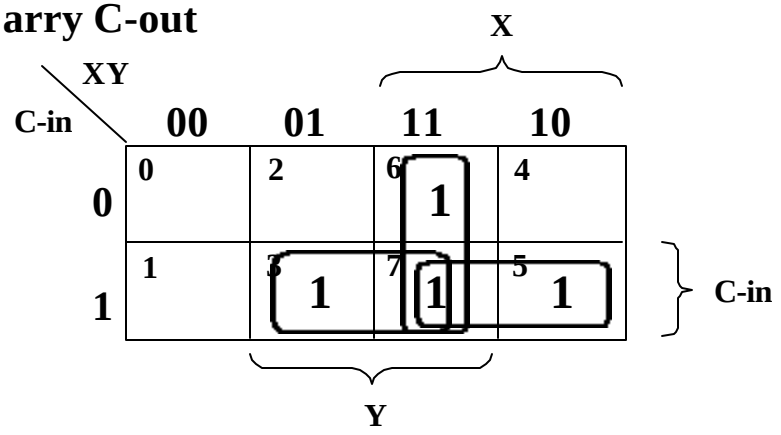
S(X,Y, C-in) = S (1,2,4,7)
C-out(x, y, C-in) = S (3,5,6,7)

Sum S



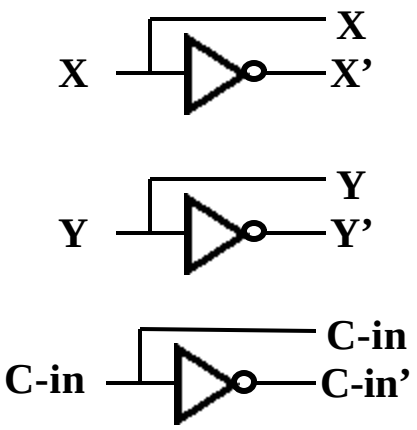
S = X'Y'(C-in) + XY'(C-in)' + XY'(C-in)' + XY(C-in)
S = X ⊕ Y ⊕ (C-in)

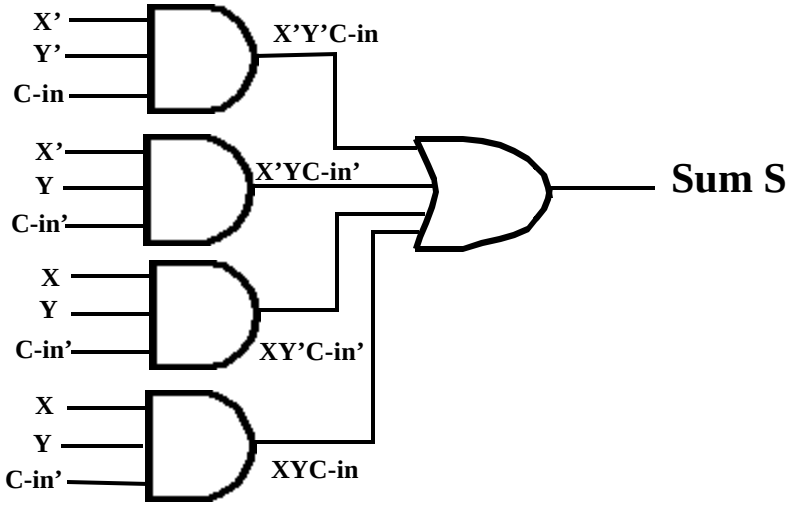
Carry C-out

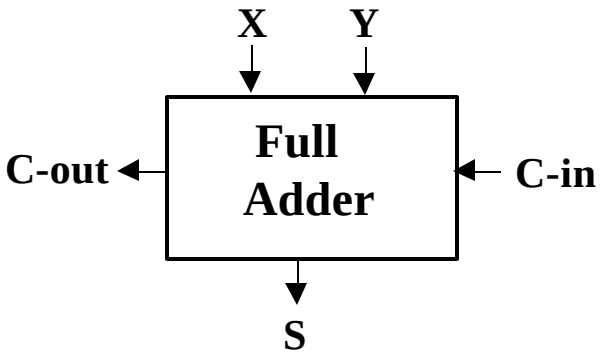


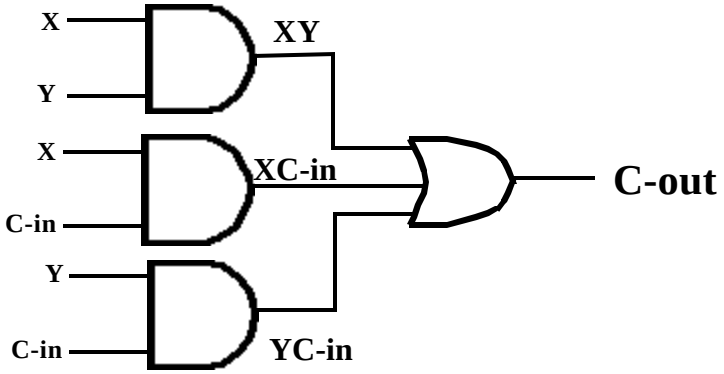
C-out = XY + X(C-in) + Y(C-in)

Full Adder Circuit Using AND-OR









n-bit Carry Ripple Adders

- An n-bit adder used to add two n-bit binary numbers can built by connecting in series n full adders.
  - Each full adder represents a bit position j (from 0 to n-1).
  - Each carry out C-out from a full adder at position j is connected to the carry in C-in of the full adder at the higher position j+1.
- The output of a full adder at position j is given by:

$$S_j = X_j \text{ Å } Y_j \text{ Å } C_j$$

$$C_{j+1} = X_j \,.\, Y_j + X_j \,.\, C_j + Y \,.\, C_j$$

- In the expression of the sum C_j must be generated by the full adder at the lower position j-1.
- The propagation delay in each full adder to produce the carry is equal to two gate delays = 2 D
- Since the generation of the sum requires the propagation of the carry from the lowest position to the highest position , the total propagation delay of the adder is approximately:

Total Propagation delay = 2 nD

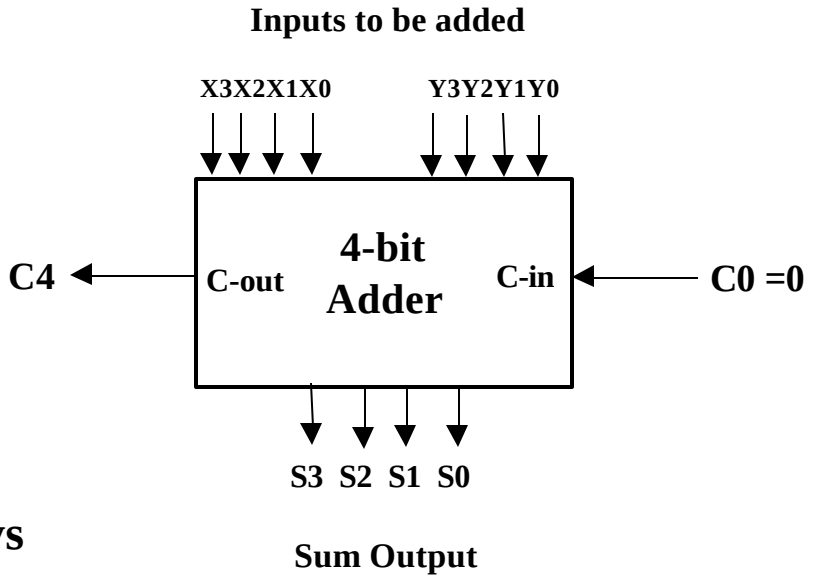
4-bit Carry Ripple Adder

Adds two 4-bit numbers:

X = X3 X2 X1 X0

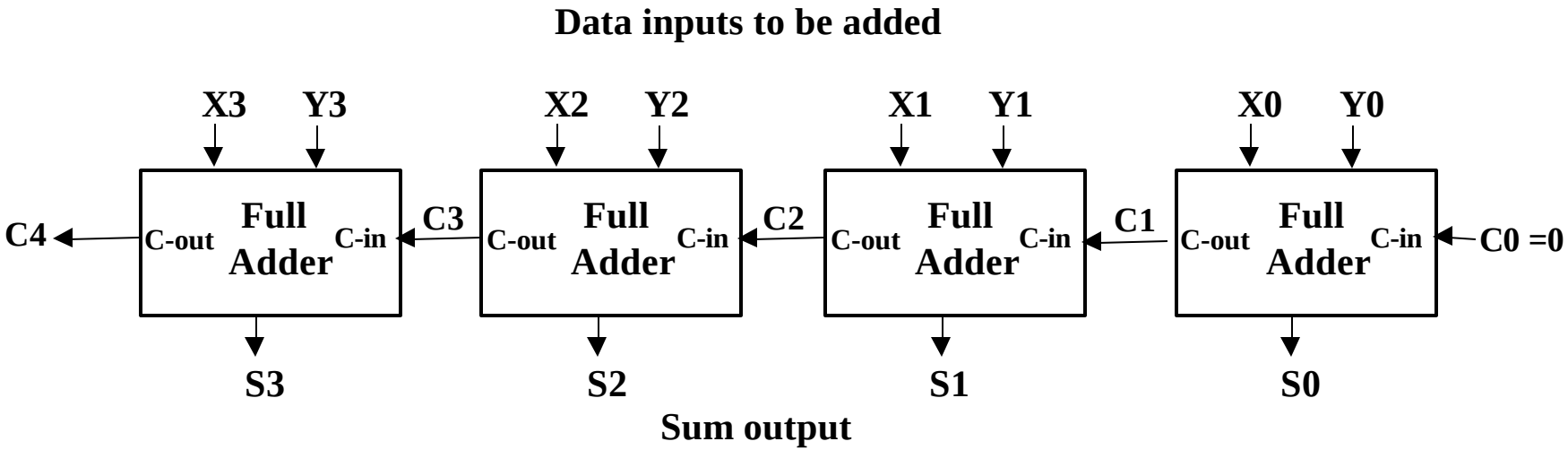
Y = Y3 Y2 Y1 Y0

producing the sum S = S3 S2 S1 S0 , C-out = C4 from the most significant position j=3



Total Propagation delay = 2 nD = 8D

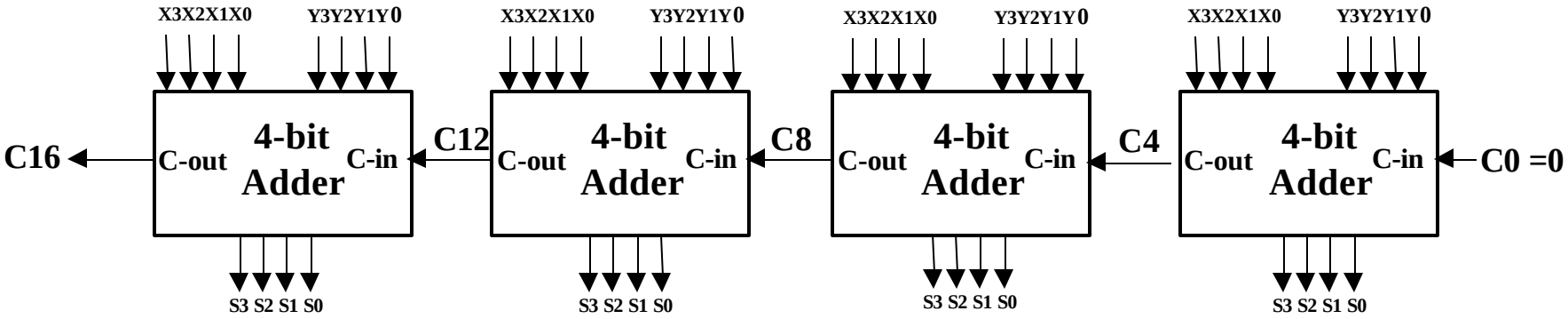
or 8 gate delays



Larger Adders

- **Example: 16-bit adder using 4, 4-bit adders**
- **Adds two 16-bit inputs X (bits X0 to X15), Y (bits Y0 to Y15) producing a 16-bit Sum S (bits S0 to S15) and a carry out C16 from most significant position.**

Data inputs to be added X (X0 to X15) , Y (Y0-Y15)



Sum output S (S0 to S15)

Propagation delay for 16-bit adder = 4 x propagation delay of 4-bit adder

= 4 x 2 nD = 4 x 8D = 32 D

or 32 gate delays

Carry Look-Ahead Adders

- The disadvantage of the ripple carry adder is that the propagation delay of adder ($2nD$) increases as the size of the adder, n is increased due to the carry ripple through all the full adders.
- Carry look-ahead adders use a different method to create the needed carry bits for each full adder with a lower constant delay equal to three gate delays.
- The carry out C-out from the full adder at position i or C_{j+1} is given by:

$$\text{C-out} = C_{i+1} = X_i \cdot Y_i + (X_i + Y_i) \cdot C_i$$

- By defining:
  - $G_i = X_i \cdot Y_i$ as the carry generate function for position i (one gate delay)
    (If $G_i = 1$ C_{i+1} will be generated regardless of the value C_i)
  - $P_i = X_i + Y_i$ as the carry propagate function for position i (one gate delay)
    (If $P_i = 1$ C_i will be propagated to C_{i+1})
- By using the carry generate function G_i and carry propagate function P_i, then C_{i+1} can be written as:

$$\text{C-out} = C_{i+1} = G_i + P_i \cdot C_i$$

- To eliminate carry ripple the term C_i is recursively expanded and by multiplying out, we obtain a 2-level AND-OR expression for each C_{i+1}

Carry Look-Ahead Adders

- For a 4-bit carry look-ahead adder the expanded expressions for all carry bits are given by:

$$C_1 = G_0 + P_0.C_0$$

$$C_2 = G_1 + P_1.C_1 = G_1 + P1.G_0 + P_1.P_0.C_0$$

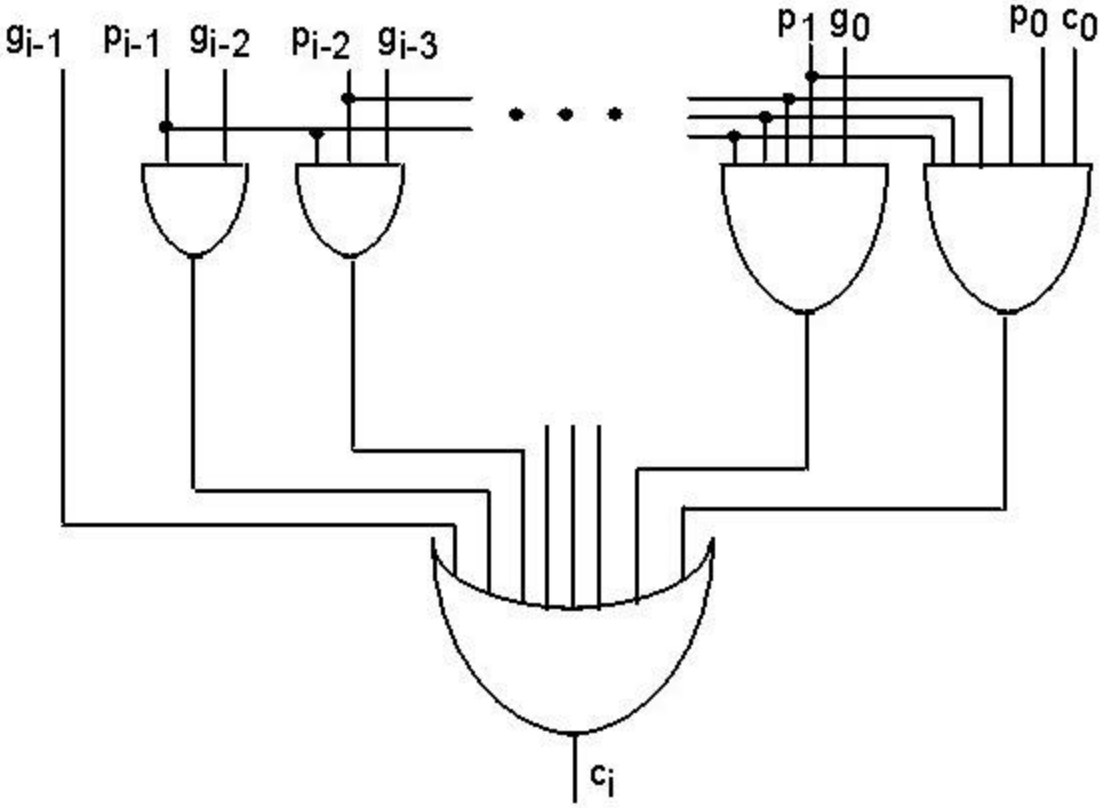
$$C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.C_0$$

$$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3 . P_2.P_1.G_0 + P_3.P_2.P_1.P_0.C_0$$

where $G_i = X_i . Y_i$ $P_i = X_i + Y_i$

- The additional circuits needed to realize the expressions are usually referred to as the carry look-ahead logic.
- Using carry-ahead logic all carry bits are available after three gate delays regardless of the size of the adder.

Carry Look-Ahead Circuit



$$C_i = G_{i-1} + P_{i-1}.G_{i-2} + \ldots + P_{i-1}.P_{i-2}.\ldots P_1 . G_0 + P_{i-1}.P_{i-2}.\ldots P_0 . C_0$$

Binary Arithmetic Operations Subtraction

- Two binary numbers are subtracted by subtracting each pair of bits together with borrowing, where needed.
- Subtraction Example:

		0 0 1 1 1 1 1 0 0	Borrow
X	229	1 1 1 0 0 1 0 1	
Y -	46	- 0 0 1 0 1 1 1 0	
	183	1 0 1 1 0 1 1 1	

Full Subtractor

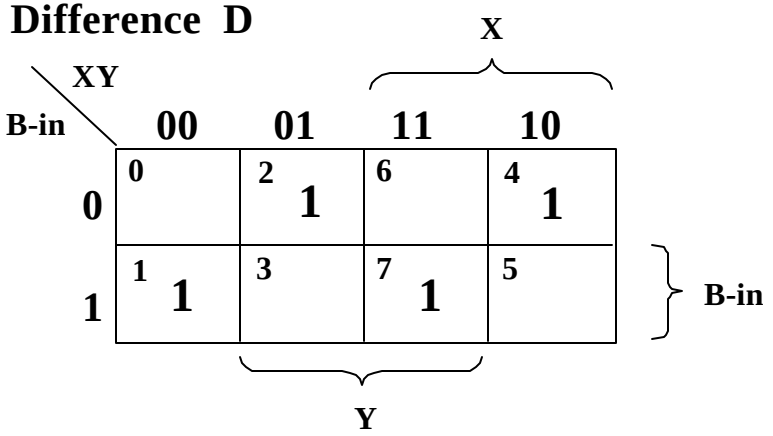
- Subtracting two single-bit binary values, Y, B-in from a single-bit value X produces a difference bit D and a borrow out B-out bit. This is called full subtraction.

Full Subtractor Truth Table

Inputs			Outputs	
X	Y	B-in	D	B-out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

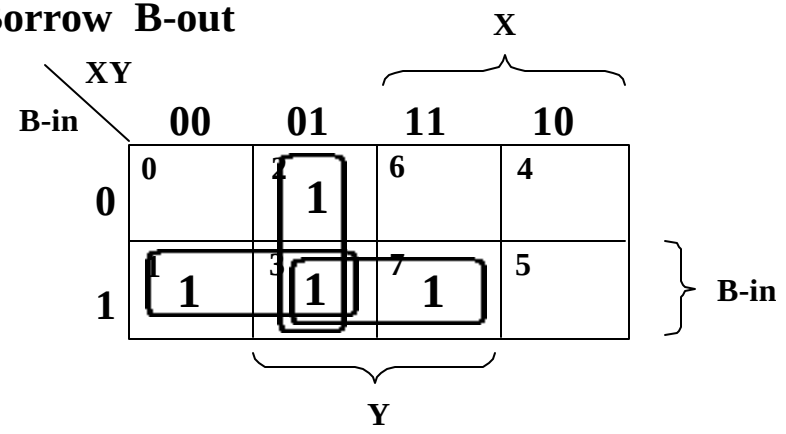
S(X,Y, C-in) = S (1,2,4,7)
C-out(x, y, C-in) = S (1,2,3,7)

Difference D



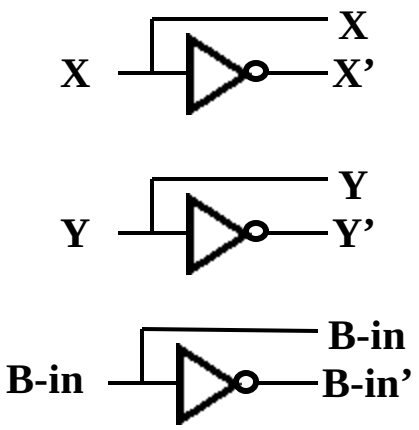
S = X'Y'(B-in) + XY'(B-in)' + XY'(B-in)' + XY(B-in)
S = X ⊕ Y ⊕ (C-in)

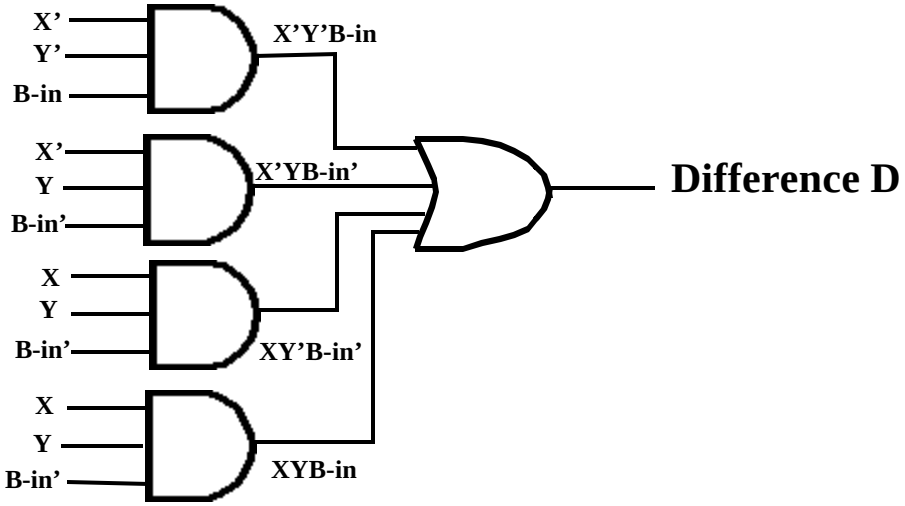
Borrow B-out

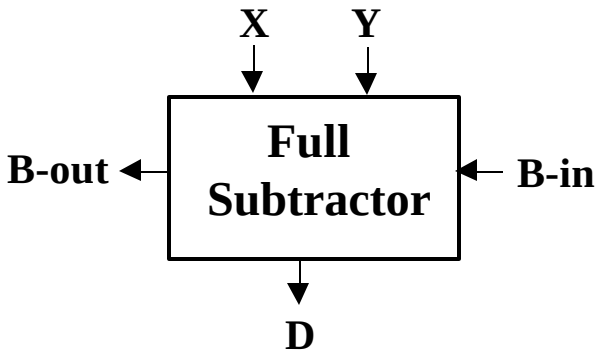


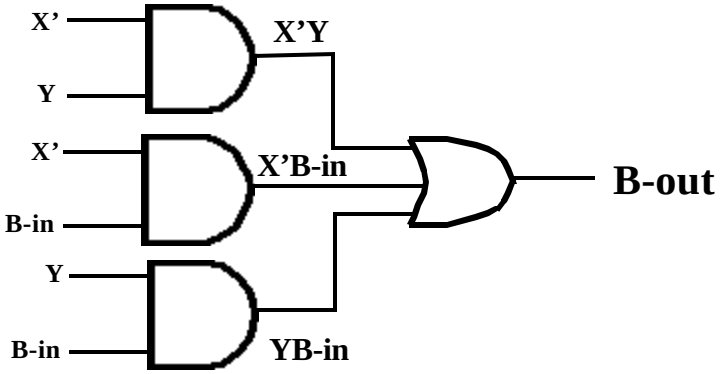
B-out = X'Y + X'(B-in) + Y(B-in)

Full Subtractor Circuit Using AND-OR









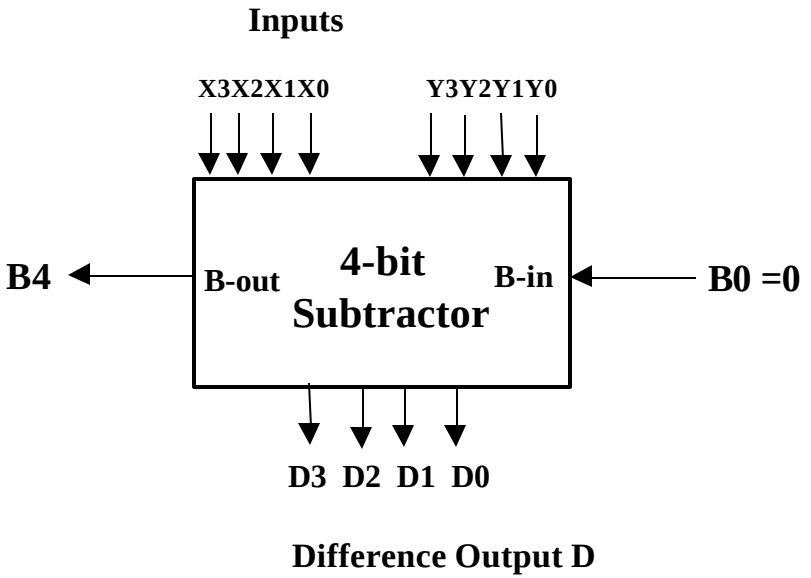
n-bit Subtractors

An n-bit subtracor used to subtract an n-bit number Y from another n-bit number X (i.e X-Y) can be built in one of two ways:

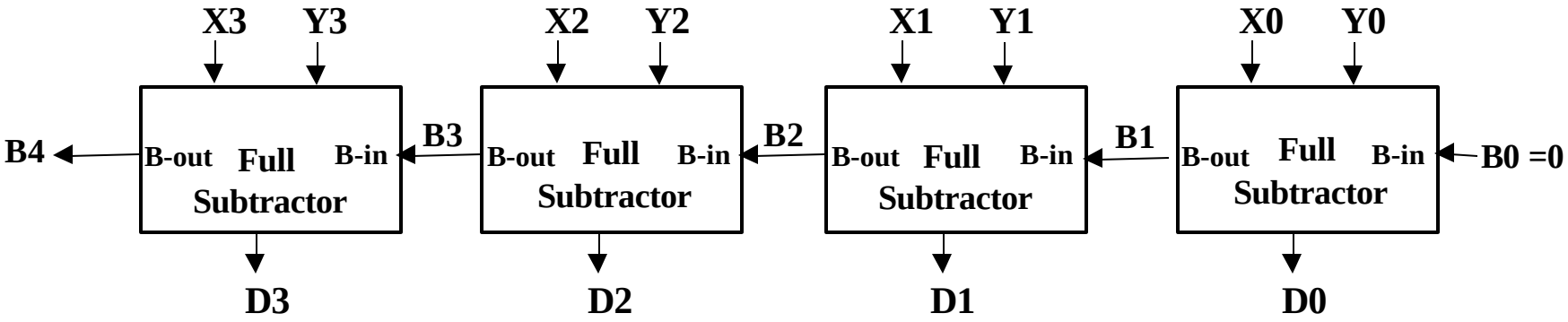
- **By using n full subtractors and connecting them in series, creating a borrow ripple subtractor:**
  - **Each borrow out B-out from a full subtractor at position j is connected to the borrow in B-in of the full subtracor at the higher position j+1.**
- **By using an n-bit adder and n inverters:**
  - **Find two's complement of Y by:**
    - **Inverting all the bits of Y using the n inverters.**
    - **Adding 1 by setting the carry in of the least significant position to 1**
  - **The original subtraction (X - Y) now becomes an addition of X to two's complement of Y using the n-bit adder.**

4-bit Borrow Ripple Subtractor

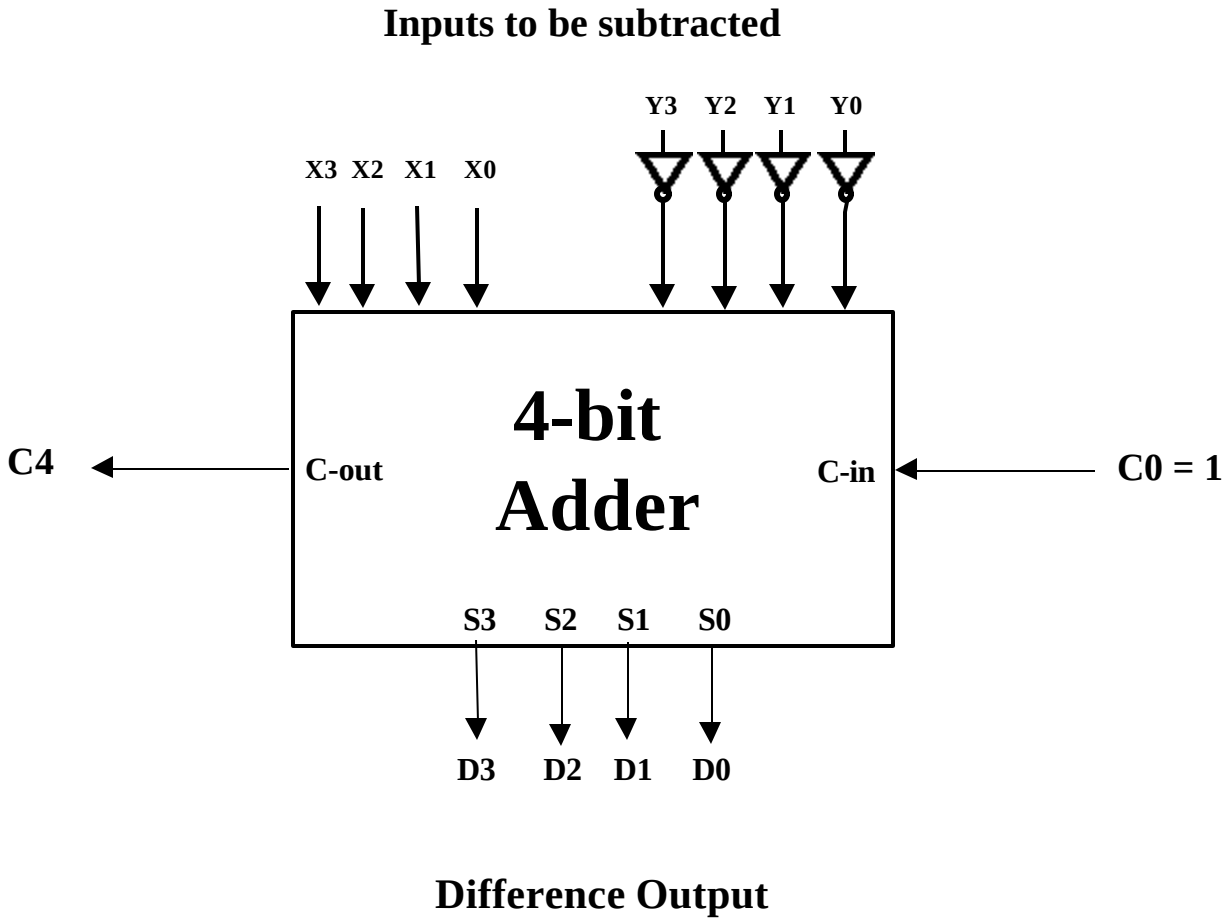
Subtracts two 4-bit numbers:
Y = Y3 Y2 Y1 Y0 from
X = X3 X2 X1 X0
Y = Y3 Y2 Y1 Y0
producing the difference D = D3 D2 D1 D0 ,
B-out = B4 from the most significant position j=3



Data inputs to be subtracted



4-bit Subtractor Using 4-bit Adder



Binary Multiplication

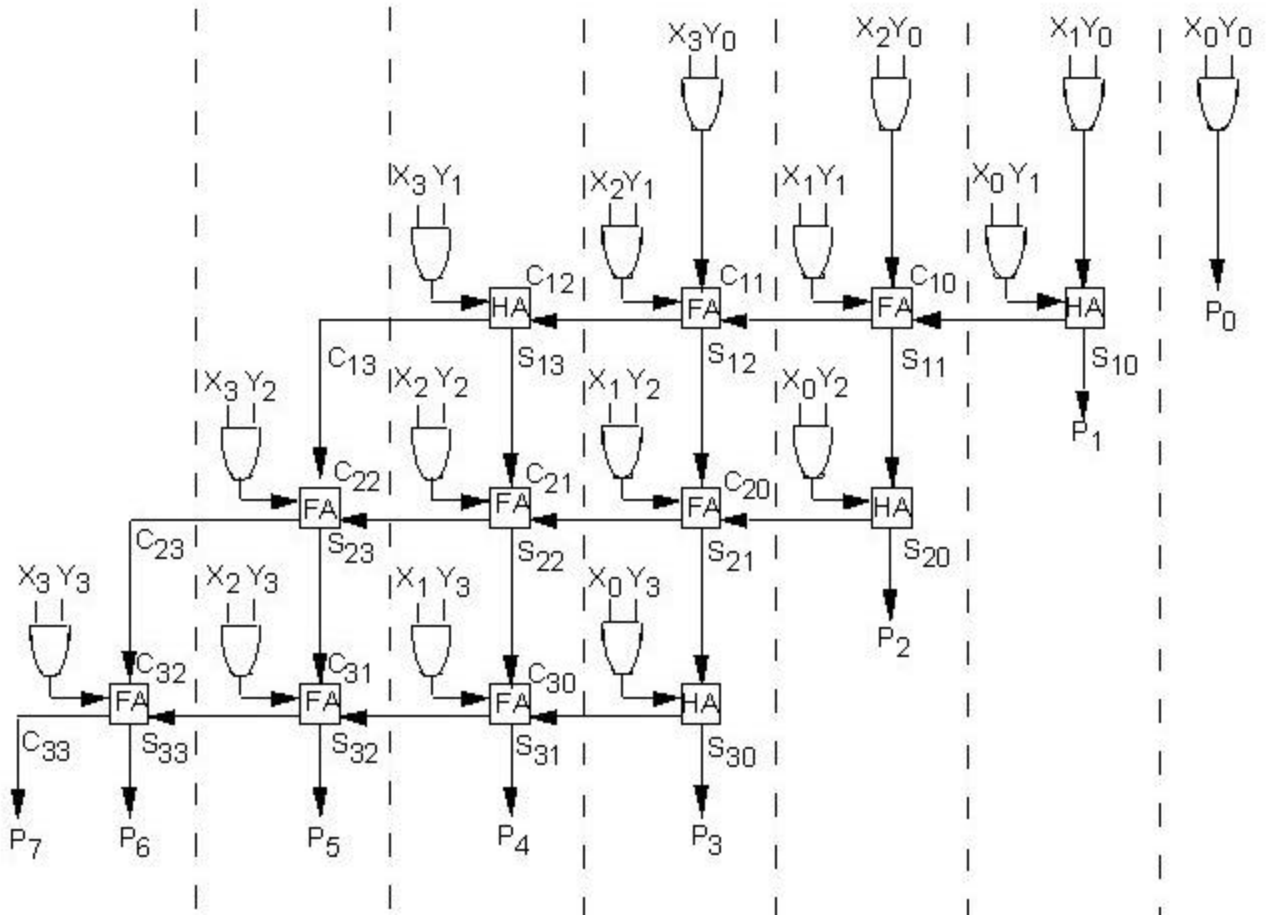
- **Multiplication is achieved by adding a list of shifted multiplicands according to the digits of the multiplier.**
- **Ex. (unsigned)**

```
   11                1 0 1 1     multiplicand (4 bits)
X 13              X   1 1 0 1   multiplier  (4 bits)
--------          ----------------
   33               1 0 1  1
   11             0 0 0 0
  _____         1 0 1 1
  143            1 0 1 1
                 ---------------------
                 1 0 0 0 1 1 1 1     Product (8 bits)
```

```
                                                        X3    X2    X1    X0
                                                  x     Y3    Y2    Y1    Y0
                                                  ---------------------------
                                                  X3.Y0 X2.Y0 X1.Y0 X0.Y0
                                            X3.Y1 X2.Y1 X1.Y1 X0.Y1
                                      X3.Y2 X2.Y2 X1.Y2 X0.Y2
                                X3.Y3 X2.Y3 X1.Y3 X0.Y3
-------------------------------------------------------------------------------
P7    P6    P5    P4    P3    P2    P1    P0
```

- **An n-bit X n-bit multiplier can be realized in combinational circuitry by using an array of n-1 n-bit adders where is adder is shifted by one position.**
- **For each adder one input is the multiplied by 0 or 1 (using AND gates) depending on the multiplier bit, the other input is n partial product bits.**

4x4 Array Multiplier

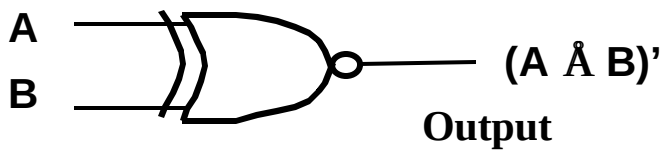


Combinational Comparators

- **Comparing two binary inputs A, B each n bits for equality (i.e A = B) is a common operation in computers.**
- **A single output combinational circuit to accomplish this can be constructed using n 2-input XNOR gates for bit-wise comparison plus one n-input AND gate. The output = 1 if A = B**
- **This can also be done by subtraction (A - B) and checking for a zero result using a single n-input NOR gate.**
- **Example: 1-bit comparator: A, B 1-bit each.**
  - **The 1-bit comparison requires a single XNOR gate**

Truth table:

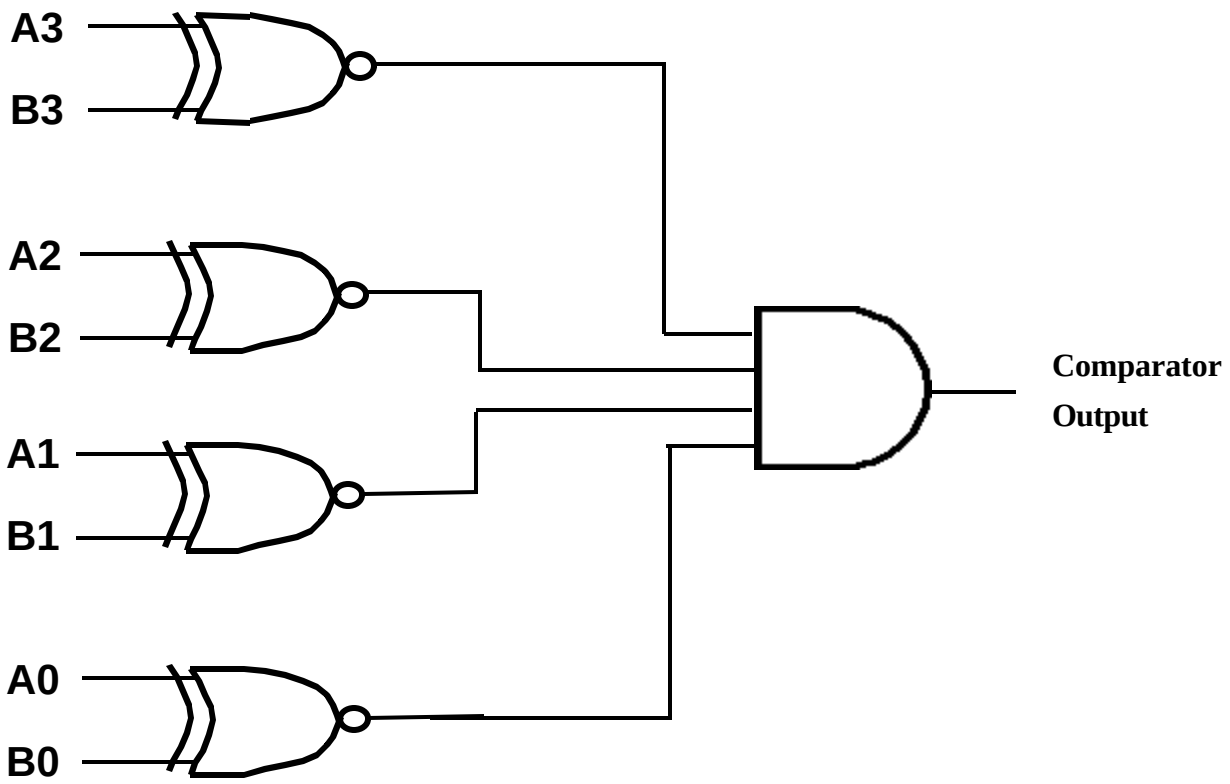
A	B	Output $(A \oplus B)'$
0	0	1
0	1	0
1	0	0
1	1	1



1-bit comparator

Example: 4-bit Comparator

Compares A = A3 A2 A1 A0 with B = B3 B2 B1 B0
Output = 1 if A = B

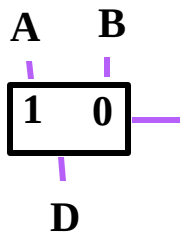


Combinational Shift Circuits

- An n-bit shift circuit (shifter) has a single n-bit data input A, and a single n-bit output R and a number of control inputs to determine the shift amount (0 to n-1).
- Possible shift operations include:
  - Shift left or right:
    - Arithmetic right shift (the sign bit is shifted in),
    - logic shift (0 is shifted in)
    - Rotate left or right.
- Example: Original data input A = 11011
    - Shift left by one : 10110
    - Logic shift right by one: 01101
    - Arithmetic shift right by one: 11101
    - Rotate left by one: 10111
- Combinational shift circuits are usually constructed using a number of levels of multiplexeres.

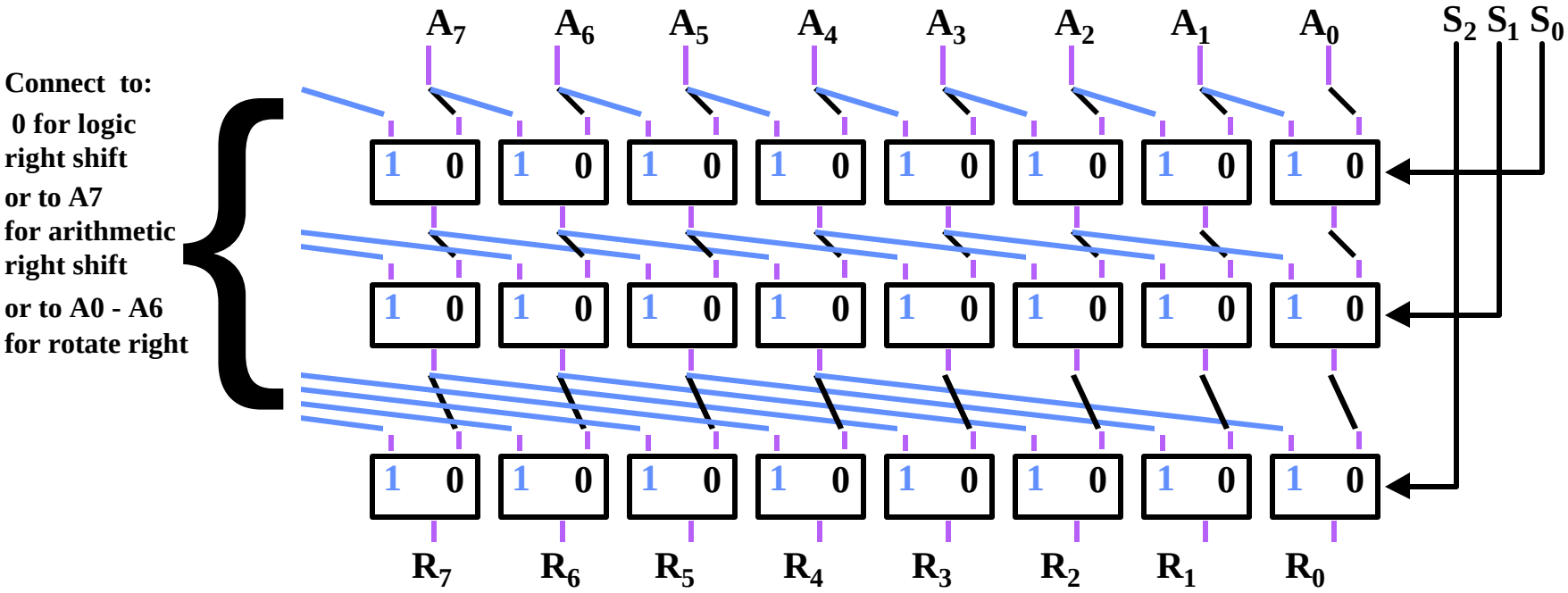
Example: Combinational 8-Bit Right Shifter

Basic Building Block 2-to-1 Mux



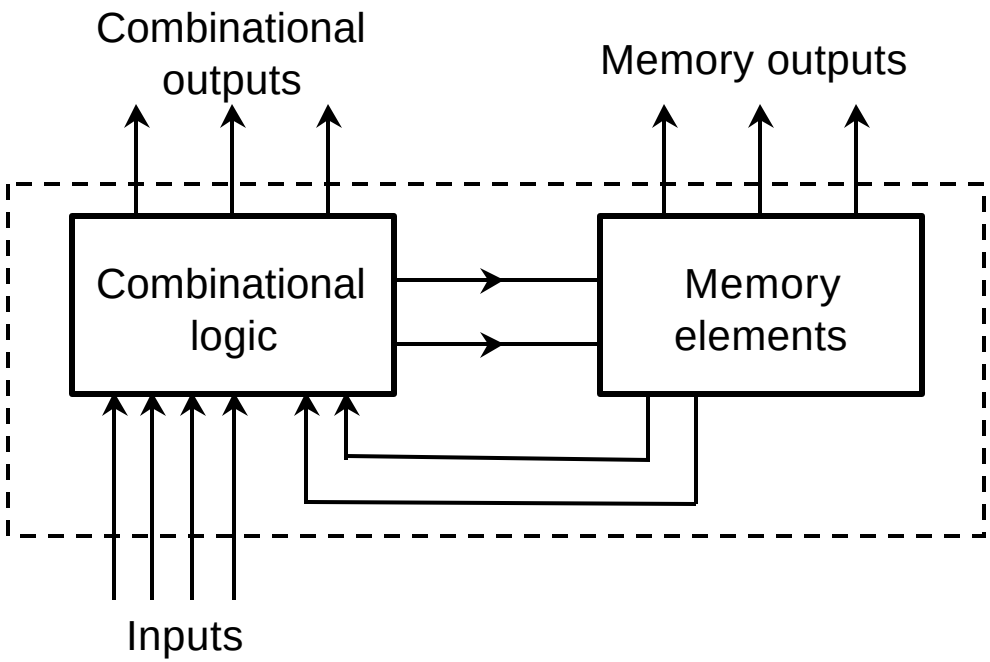
Three levels of Muxes used

S_2 S_1 S_0
shift amount from 0 to 7



- **Propagation delay: 2 gate delays per level x 3 levels = 6 gate delays**
- **How many Mux levels for 32-bit shifter? Propagation delay?**

Sequential Logic Circuits



Sequential circuit = Combinational logic + Memory Elements

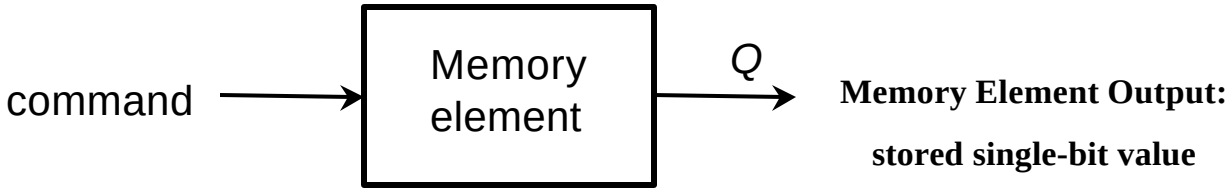
Current State of A sequential Circuit: Value stored in memory elements (value of state variables).

State transition: A change in the stored values in memory elements thus changing the sequential circuit from one state to another state.

Sequential Circuit Buliding Blocks:

Generic Memory Elements

- **A Memory Element: A logic device that can remember a single-bit value indefinitely, or change its value on command from its inputs.**



- **The output Q of the memory element represents the value stored in the memory element. This is also called the state variable of the memory elements. A memory element can be in one of two possible states:**
  - Q = 0 (the memory element has 0 stored), also said be in state 0.
  - Q =1 (the memory element has 1 stored), also said to be in state 1.
- **The commands to the memory element formed by its input(s) may include:**
  - **Set: Store 1 (Q=1) in the memory element.**
  - **Reset: Store 0 (Q=0) in the memory element.**
  - **Flip: Change stored value from 0 to 1 or from 1 to 0.**
  - **Hold value: Memory value does not change.**
- **Memory Element state transition: A change in the stored value from 0 to 1, or from 1 to 0 such as that caused by a flip command.**

Sequential Circuit Memory Elements: Latches, Flip-Flops

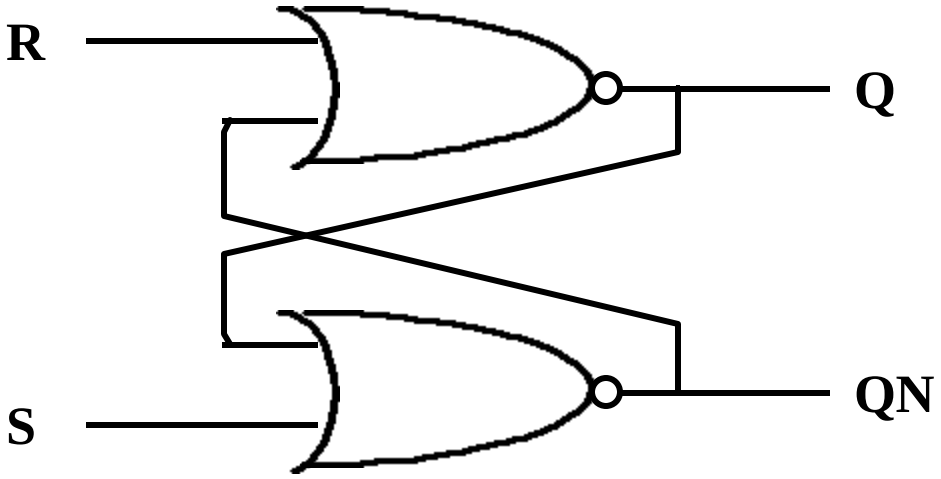
- Latches and flip-flops are the basic single-bit memory elements used to build sequential circuit with one or two inputs/outputs, designed using individual logic gates and feedback loops.
- Latches:
  - The output of a latch depends on its current inputs and on its previous inputs and its change of state can happen at any time when its inputs change.
- Flip-Flops:
  - The output of a flip-flop also depends on current and previous input but the change in output (change of state or state transition) occurs at specific times determined by a clock input.

Sequential Circuit Memory Elements: Latches, Flip-Flops

- **Latches:**
  - **S-R Latch**
  - **S-R Latch With Enable**
  - **D-Latch**
- **Flip-Flops:**
  - **Edge-Triggered D Flip-Flop**
  - **Master/Slave S-R Flip-Flop**
  - **Master/Slave J-K Flip-Flop**
  - **Edge-Triggered J-K Flip-Flop**
  - **T Flip-Flop With Enable**

S-R Latch

- **An S-R (set-reset) latch can be built using two NOR-gates forming a feedback loop.**
- **The output of the S-R latch depends on current as well as previous inputs or state, and its state (value stored) can change as soon as its inputs change.**



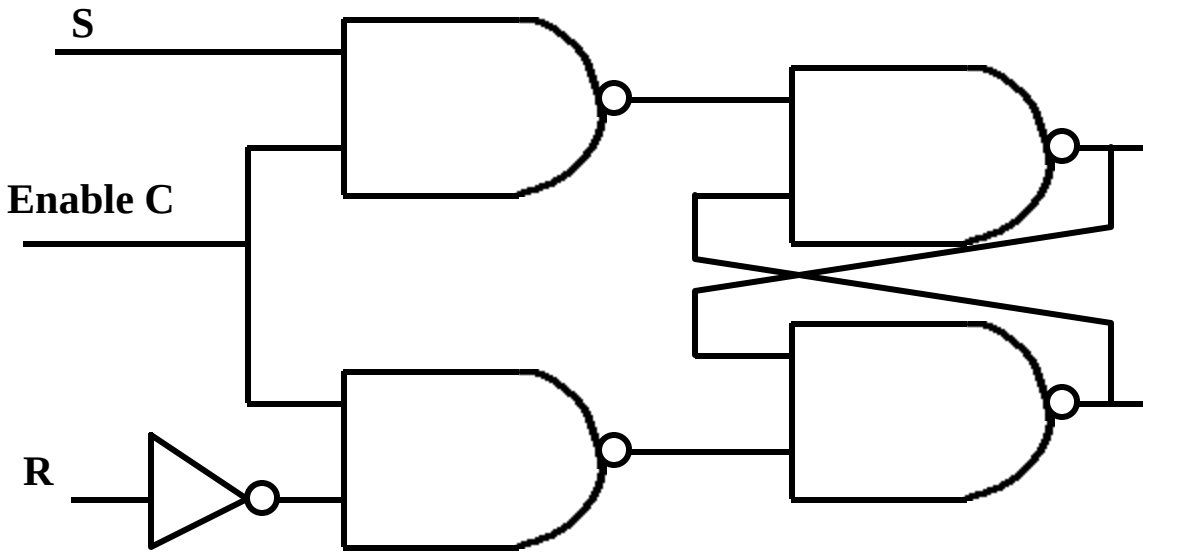
Circuit

Function Table

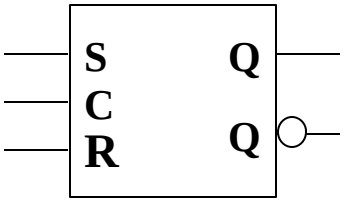
S	R	Q	QN
0	0	last Q	Last QN
0	1	0	1
1	0	1	0
1	1	0	0

S-R Latch With Enable

- Since the S-R latch is responsive to its inputs at all times an enable line C is used to disable or enable state transitions.
- Behaves similar to a regular S-R latch when enable C=1



Circuit



Logic Symbol

Function Table

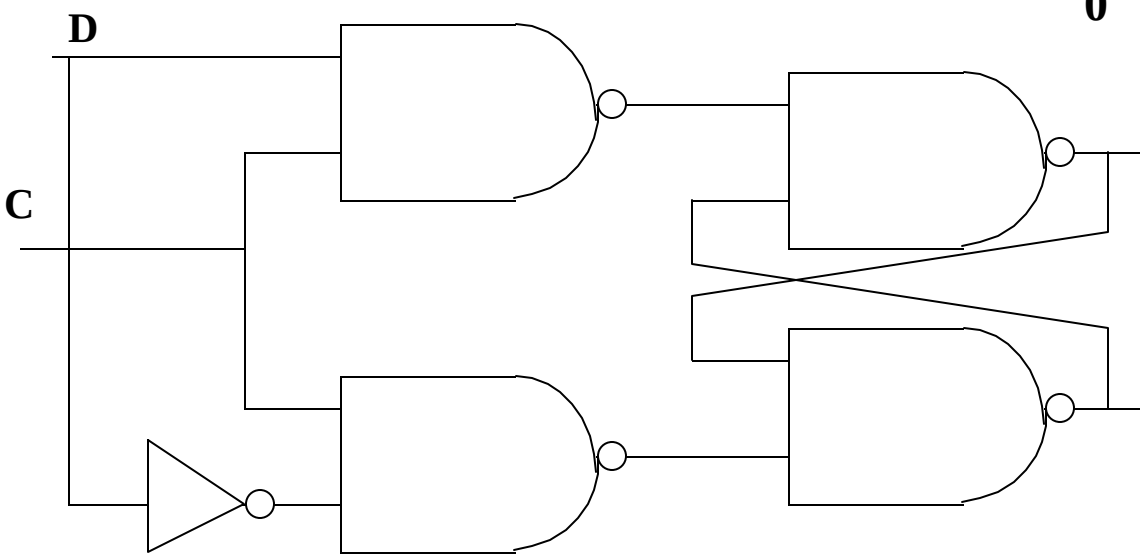
S	R	C	Q	QN
0	0	1	last Q	last QN
0	1	1	0	1
1	0	1	1	0
1	1	1	0	0
x	x	0	last Q	last QN

D-Latch

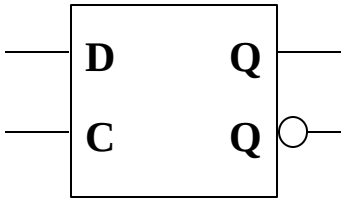
- **Similar to S-R latch with an enable line, but both S, R are generated from one input D (data) and an inverter.**
- **Stores the value of its input D when enable C =1.**

Function Table

C	D	Q	QN
1	0	0	1
1	1	1	0
0	x	Last Q	Last QN



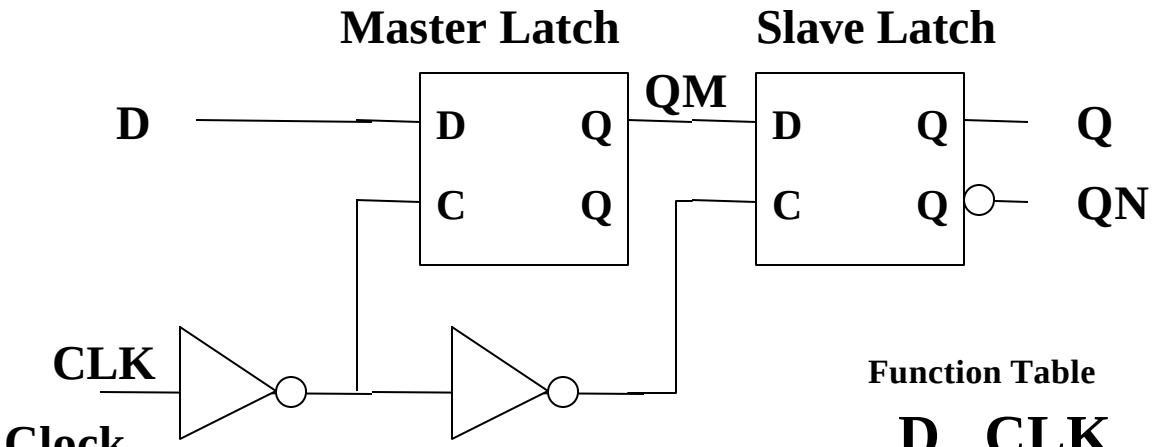
Circuit



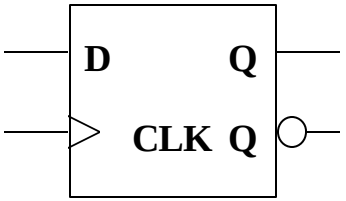
Logic Symbol

Edge-Triggered D Flip-Flop

- Uses a pair of D latches and inverters.
- Similar in behavior to a D latch except that output and state changes happen at the rising or falling edge of an input clock.
- A D Flip-Flop triggered on the rising edge of the clock is given by:



Circuit



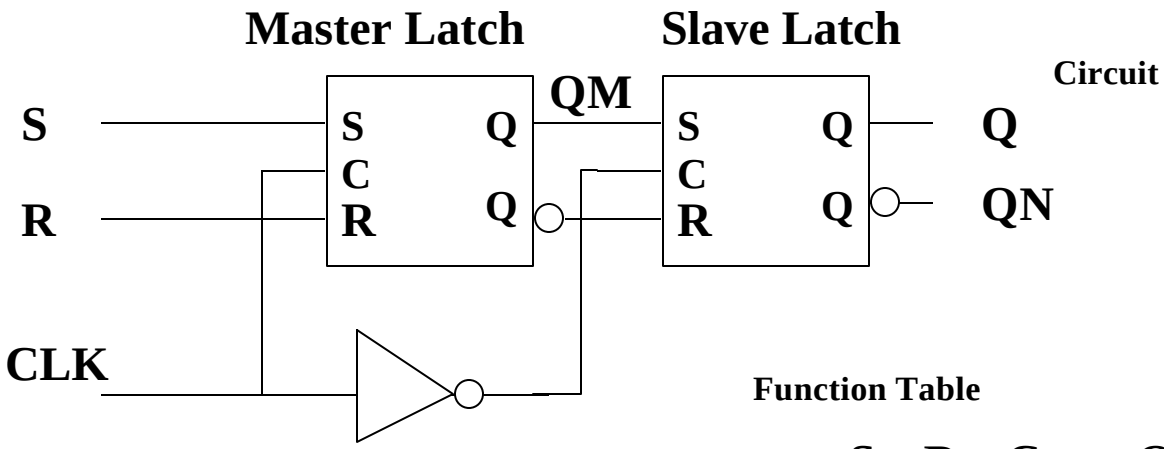
Logic Symbol

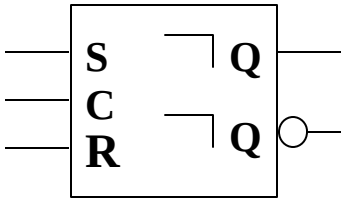
Function Table

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

Master/Slave S-R Flip-Flop

- **S-R latches are substituted for the D latches in the negative-edge triggered D flip flop**





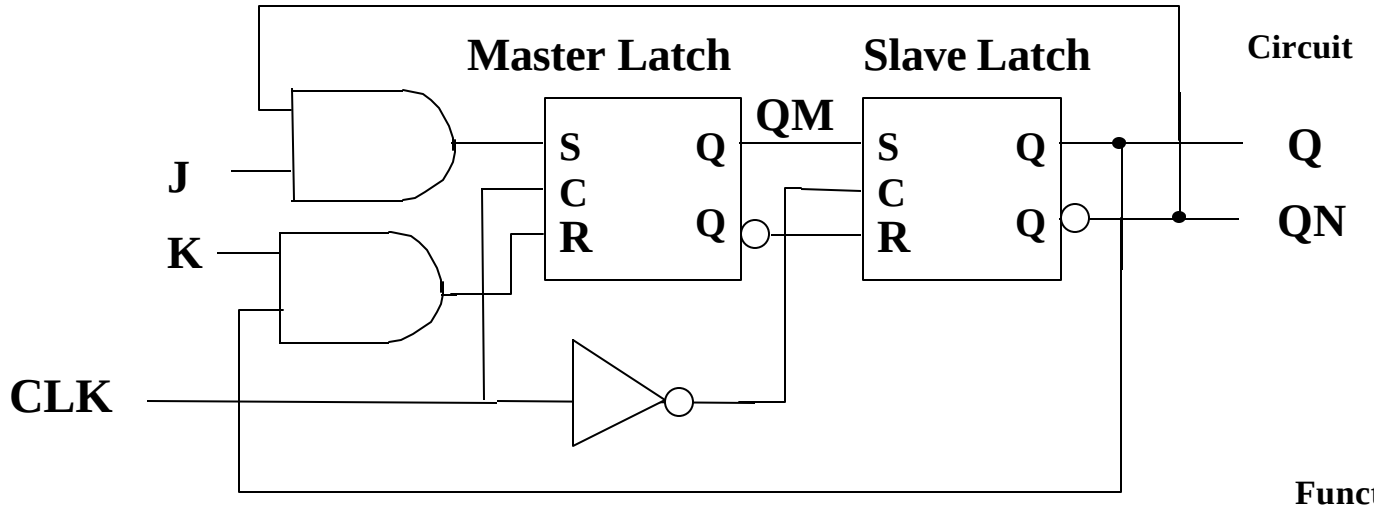
Logic Symbol

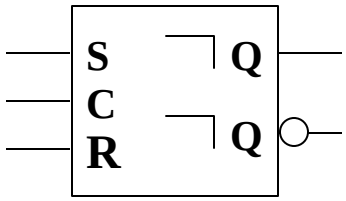
Function Table

S	R	C	Q	QN
x	x	0	last Q	last QN
0	0	⌐	last Q	last QN
0	1	⌐	0	1
1	0	⌐	1	0
1	1	⌐	undef.	undef.

Master/Slave J-K Flip-Flop

- **Solves the problem when both S=R=1**
- **When J=K=1 the last state is inverted.**





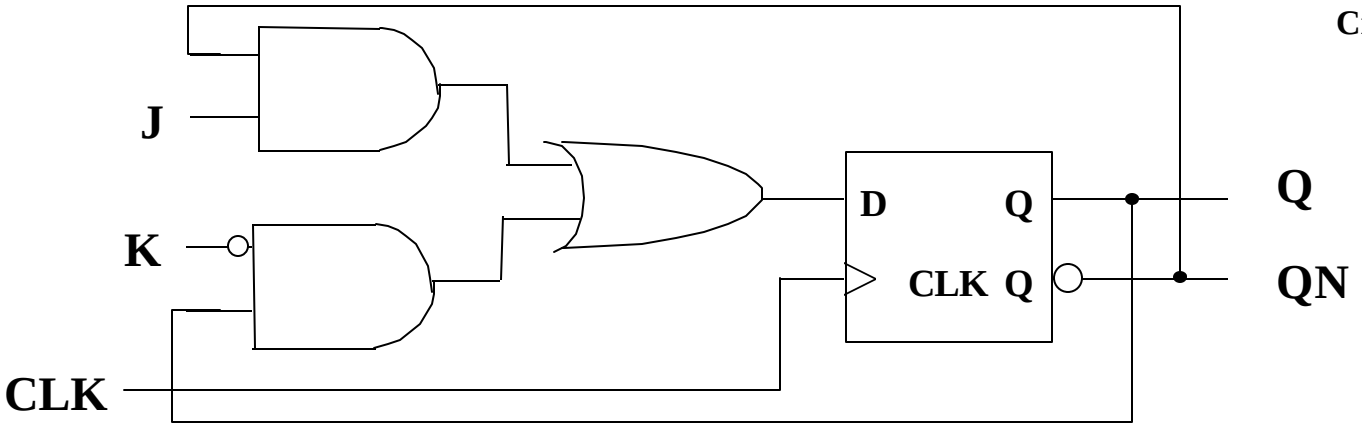
Logic Symbol

Function Table

J	K	C	Q	QN
x	x	0	last Q	last QN
0	0	⎍	last Q	last QN
0	1	⎍	0	1
1	0	⎍	1	0
1	1	⎍	last QN	last Q

Edge Triggered J-K Flip-Flop

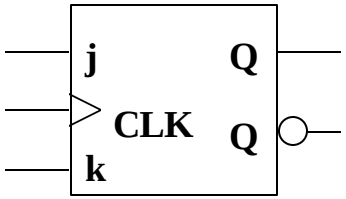
- **Created from an edge-triggered D flip-flop**



Function Table

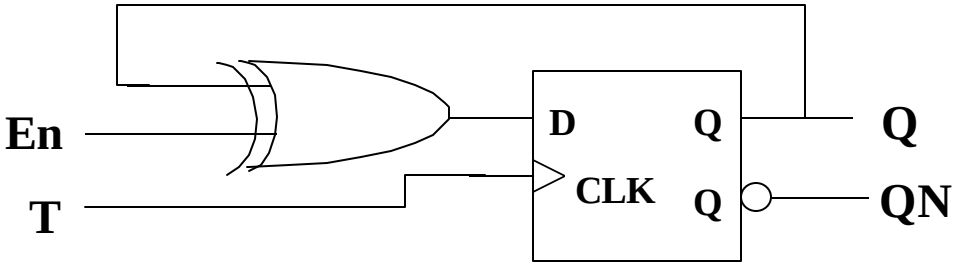
J	K	C	Q	QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN
0	0	↑	last Q	last QN
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	last QN	last Q

Logic Symbol



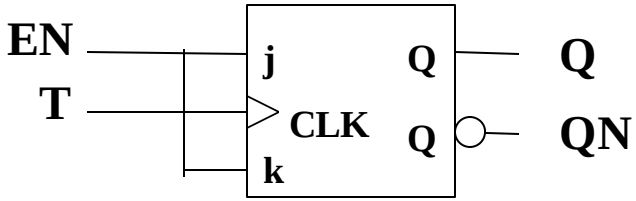
T Flip-Flop With Enable

- **Changes state on every clock cycle (rising edge of T).**



Circuit

OR



Function Table

T	En	Q	QN
x	0	last Q	last QN
↑	1	last QN	last Q

Clocked Synchronous State-Machines

- **Such machines have the characteristics:**
  - **Sequential circuits designed using flip-flops.**
  - **All flip-flops use a common clock (clocked synchronous).**
  - **A machine using n flip-flops (state memory) has n state variables (the outputs of the flip-flops) and 2^n states.**
  - **In general, the next state and output of the machine both depend on the current state of the machine and on the current input:**

Next state = F(current state, input)

output = G(current state, input)

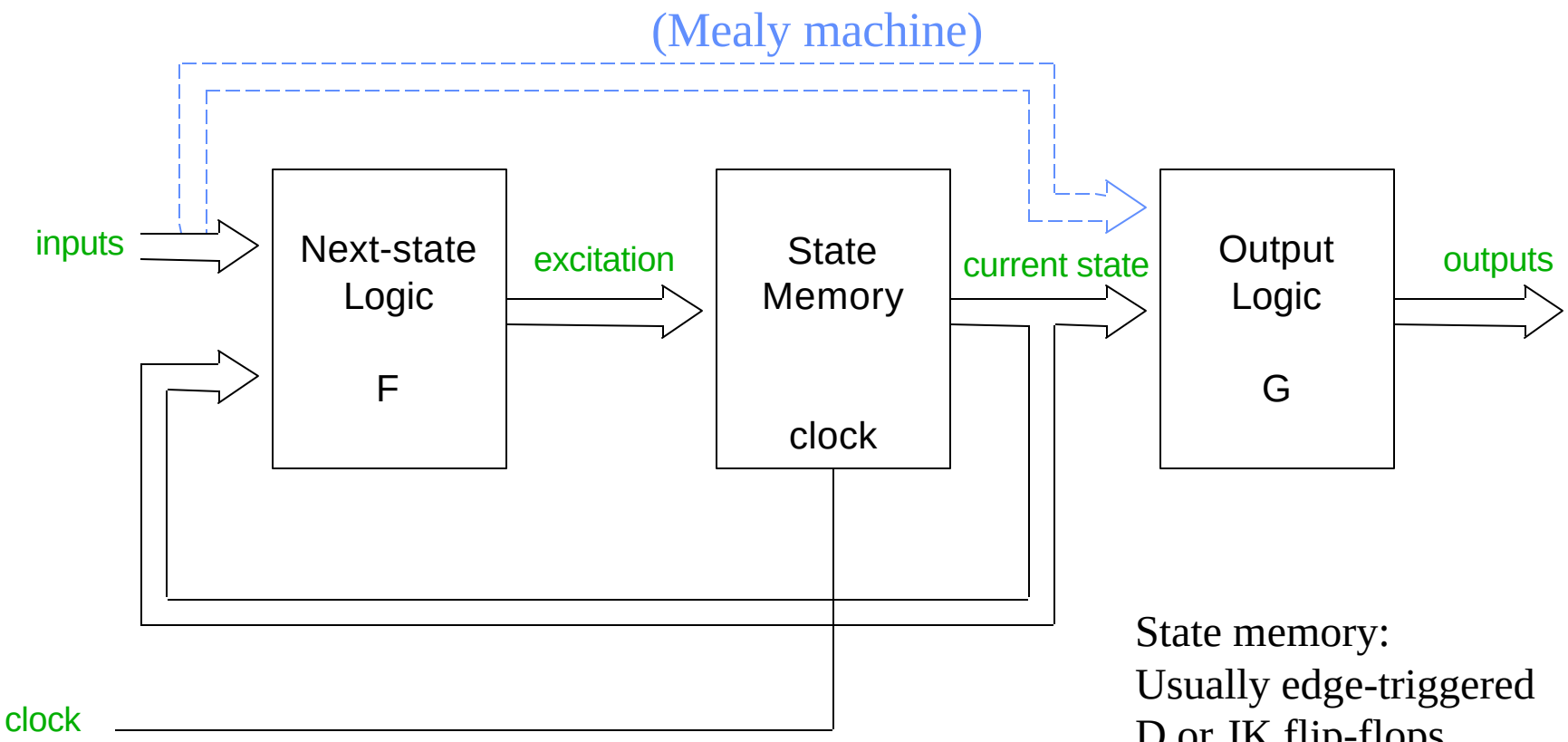
This type of state machine is called Mealy Machine

  - **In some cases the next output depends only on the current state and not directly on the current input**

Next state = F(current state, input) output = G(current state)

Such machines are called Moore machines.

Clocked Synchronous State-Machine Model



State memory:
Usually edge-triggered
D or JK flip-flops

Moore Machine

Latch/Flip-Flop Characteristic Equations

Device	Characteristic Equations
S-R latch	Q* = S+R'.Q
D latch	Q* = D
Edge-triggered D flip-flop	Q* = D
Master/Slave S-R flip-flop	Q* = S+R'.Q
Master/Slave J-K flip flop	Q* = J.Q' + K'.Q
Edge Triggered J-K flip-flop	Q* = J.Q' + K'.Q
T flip-flop	Q* = Q'
T flip-flop with enable	Q* = EN.Q' + EN'.Q

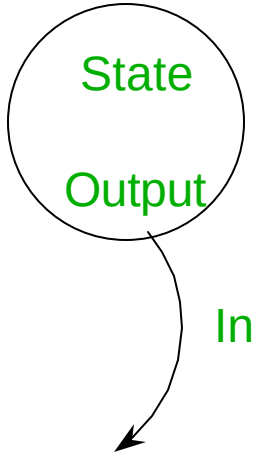
Clocked Synchronous State-machine Analysis

Given the circuit diagram of a state machine:

1. **Analyze the combinational logic to determine flip-flop input (excitation) equations:** $D_i = F_i (Q, \text{inputs})$
   - **The input to each flip-flop is based upon current state and circuit inputs.**
2. **Substitute excitation equations into flip-flop characteristic equations, giving transition equations:** $Q_i^* = H_i(D_i)$
3. **From the circuit, find output equations:** $Z = G (Q, \text{inputs})$
   - **The outputs are based upon the current state and possibly the inputs.**
4. **Construct a state transition/output table from the transition and output equations:**
   - **Similar to truth table.**
   - **Present state on the left side.**
   - **Outputs and next state for each input value on the right side.**
   - **Provide meaningful names for the states in state table, if possible.**
5. **Draw the state diagram which is the graphical representation of state table.**

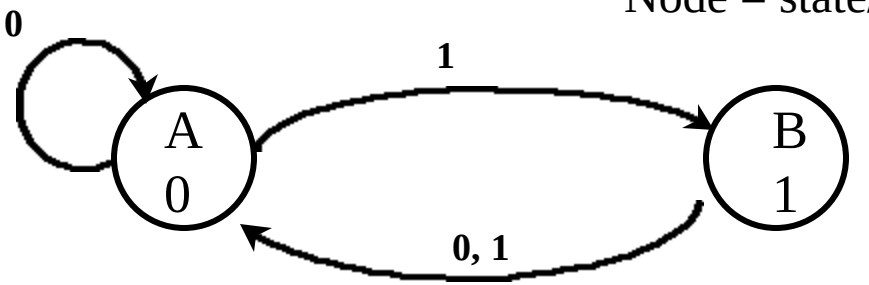
State Diagram

Basic Format:

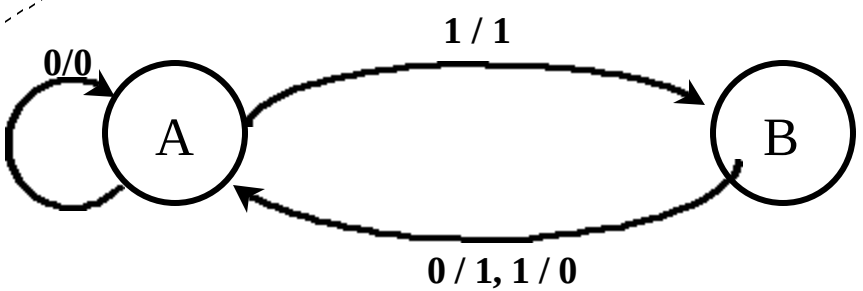


Moore

Format:
Arc = input X
Node = state/output Q



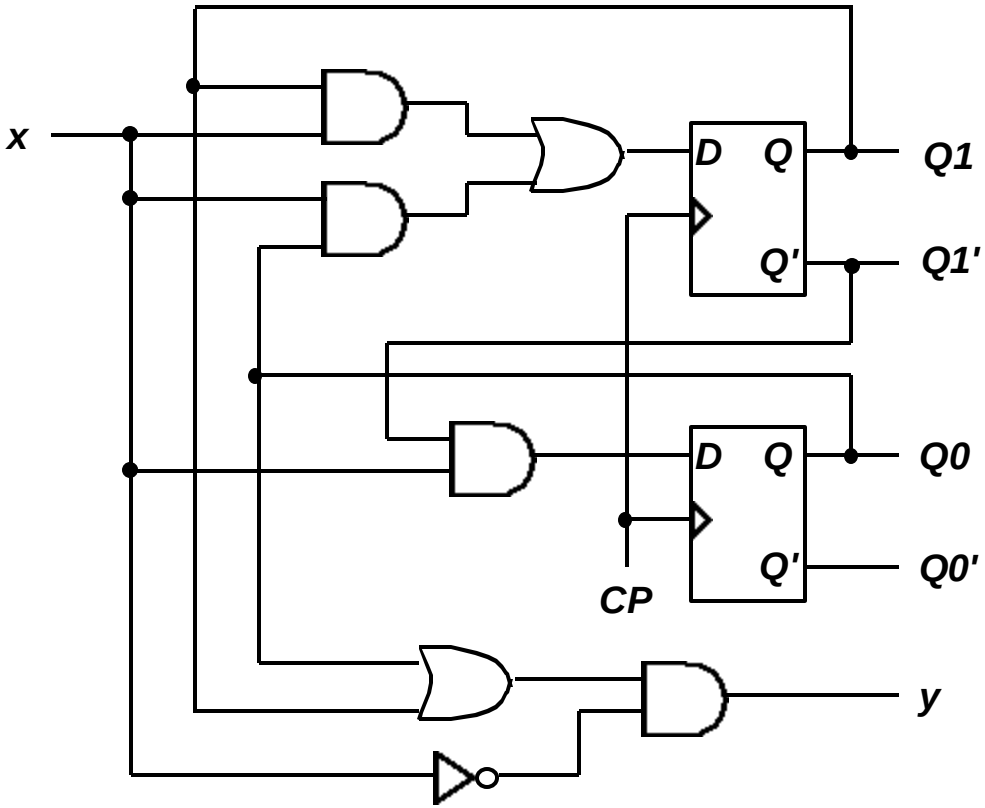
Mealy



Format:
Arc = input X / mealy output Y
Node = state

State Machine Analysis Example

Analyze the state machine:



1 Input (or excitation) equations:

D0 = Q1'. X

D1 = Q1 . x + Q0 . x

2 Characteristic equations:

Q0* = D0

Q1* = D1

Find State equations:

Q0* = Q1'. x

Q1* = Q1 . x + Q0 . x

3 Output equation:

y = (Q0 + Q1) . x'

This is a Mealy Machine since output = G(current state, input)

State Machine Analysis Example

4 **From the *state equations* and *output equation*, construct the *state transition/output table*:**

State equations:

Q0* = Q1'. x

Q1* = Q1 . x + Q0 . x

Output equation:

y = (Q0 + Q1) . x'

		x = 0	x = 1
Q1	Q0	Q1* Q0* , y	
0	0	00,0	01,0
0	1	00,1	11,0
1	0	00,1	10,0
1	1	00,1	10,0

Input

Current State

Next State when x =0

Output for current state when x =0

Next State when x =1

Output for current state when x =1

State Machine Analysis Example

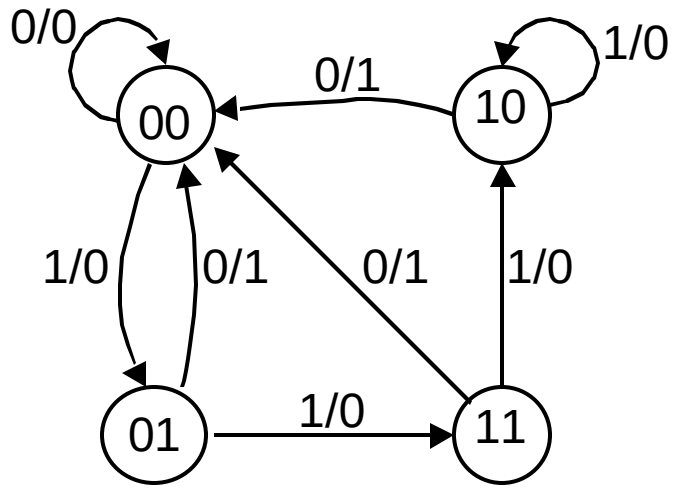
5 **Draw the state diagram of the state machine.**

state transition/output table

Q1	Q0	x = 0	x = 1
0	0	00,0	01,0
0	1	00,1	11,0
1	0	00,1	10,0
1	1	00,1	10,0

Q1* Q0* , y

state diagram



Arc = input x / output y
Node = state

Clocked State-machine Analysis: State Naming

- **State Naming:**
  - Optionally name the states and substitute state names S for state-variable combinations in transition/output table and in state diagram.
  - Example: For a circuit with two flip-flops:

Q1	Q0	State Name
0	0	A
0	1	B
1	0	C
1	1	D

Clocked State-machine Analysis Example: Transition/Output Table Using State Names

For the last example naming The States:

Q1	Q0	State Name
0	0	A
0	1	B
1	0	C
1	1	D

Transition/output Table:

	Q1	Q0	x = 0	x = 1
A	0	0	00,0	01,0
B	0	1	00,1	11,0
C	1	0	00,1	10,0
D	1	1	00,1	10,0

Q1* Q0* , y

Transition/output Table using state names:

S	x = 0	x = 1
A	A,0	B,0
B	A,1	D,0
C	A,1	C,0
D	A,1	C,0

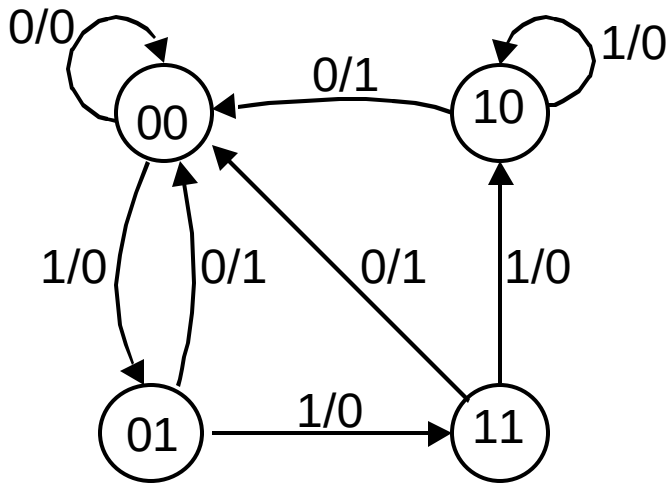
S* , y

Clocked State-machine Analysis Example: State Diagram Using State Naming

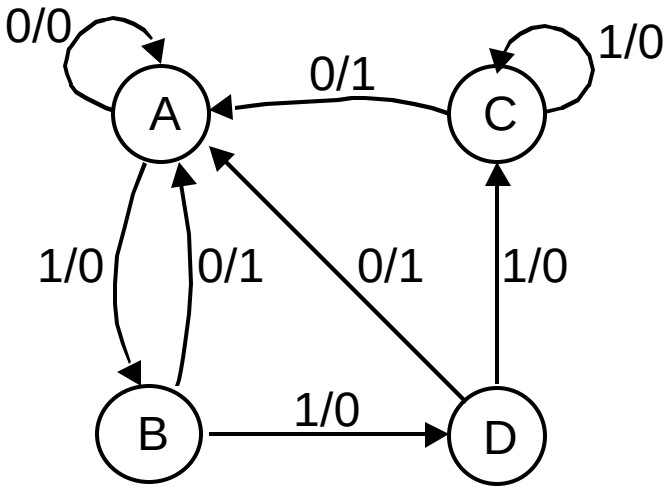
Naming The States:

Q1	Q0	State Name
0	0	A
0	1	B
1	0	C
1	1	D

State Diagram without state naming:



State Diagram with state naming:

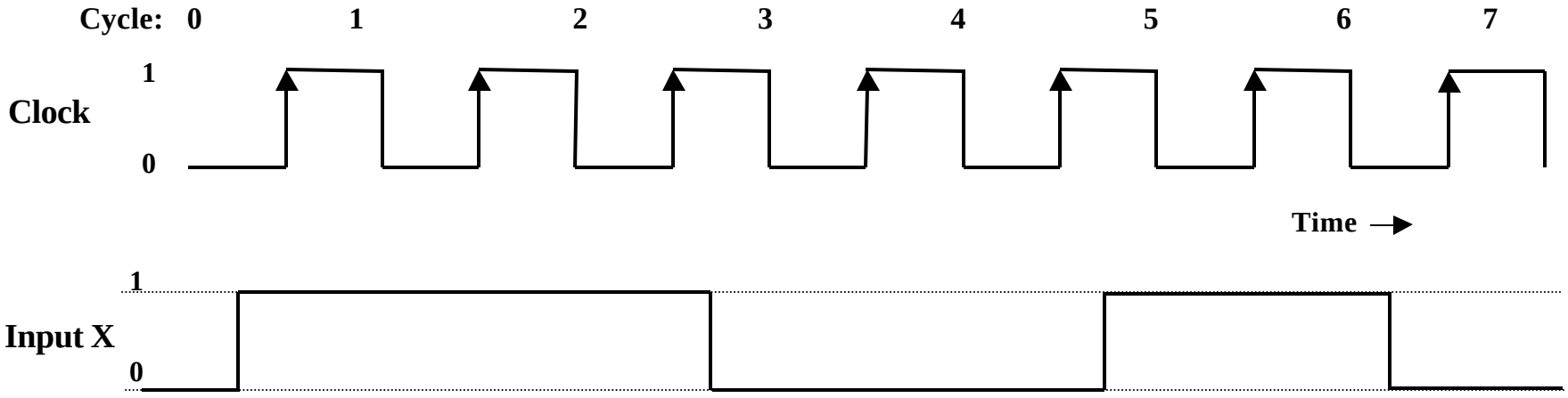


Arc = input x / output y
Node = state

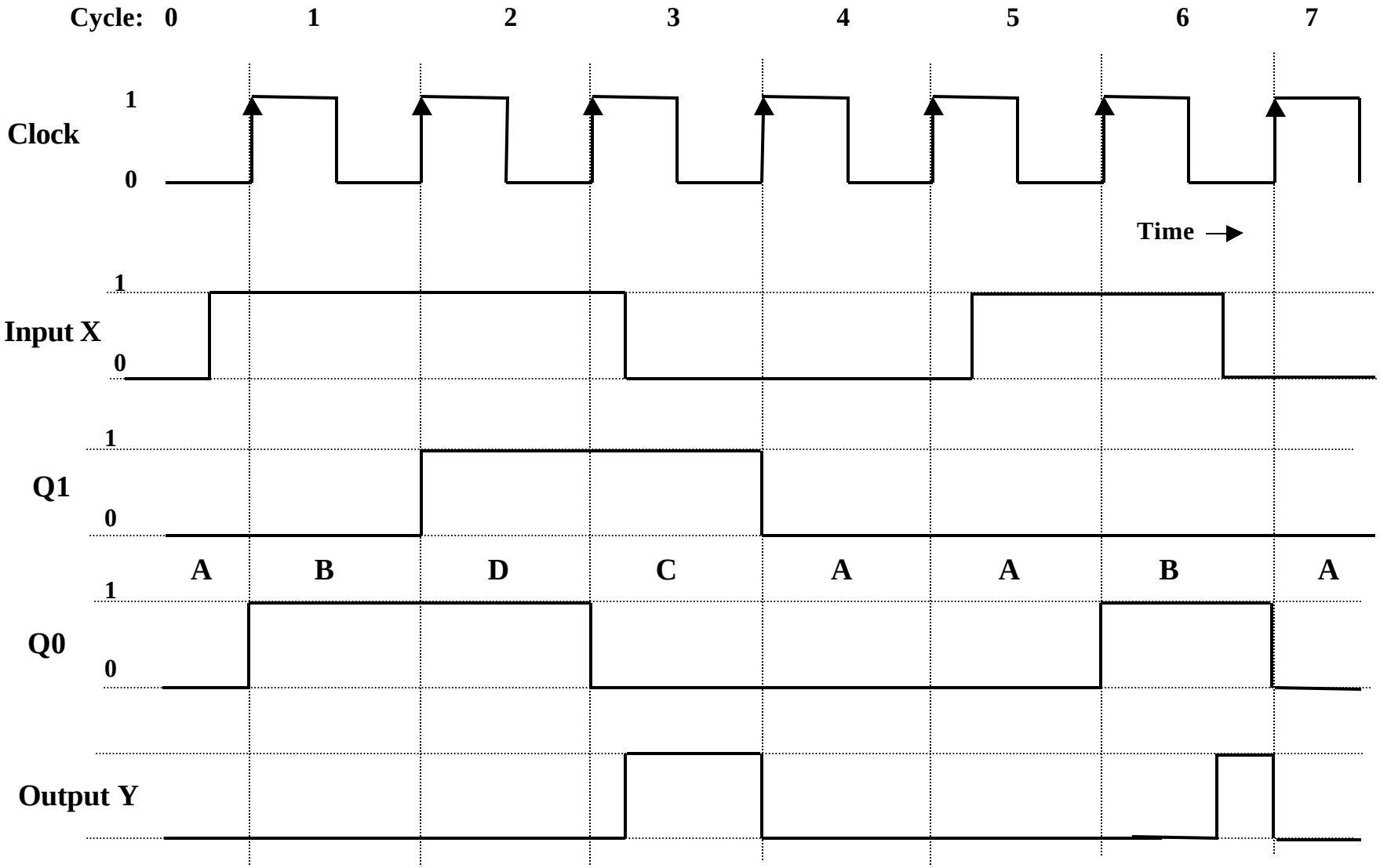
Clocked State-machine Analysis: State Machine Timing Diagram

- The timing diagram for a state machine graphically shows the state machine response in terms of state variables and output signals vs. time for given time-varying input signals and a given initial state.
- State machine timing diagrams can be generated using transition/output tables or state diagrams.
- Timing diagrams can be used to account for both combinational and flip-flop propagation delays.

- Example: For the state machine in the previous example show the timing diagram for the following input, assuming an initial state A and ignoring propagation delays:

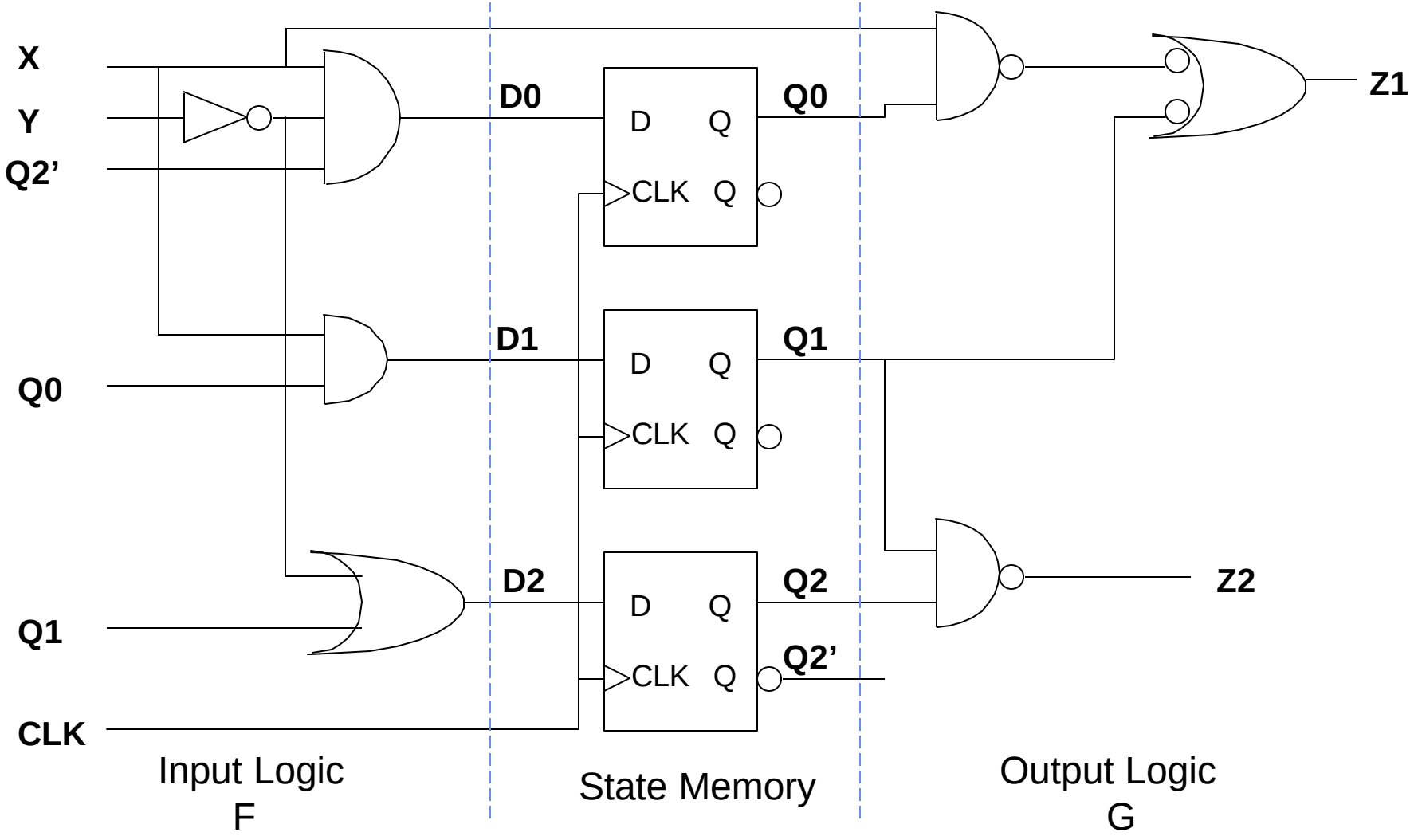


State Machine Timing Diagram Example



State Machine Analysis Example 2

Analyze the state machine:



State Machine Analysis Example 2

Excitation Equations

D0 = X . Y'.Q2

D1 = X . Q0

D2 = Y' + Q1

1

Characteristic Equations

Q0* = D0

Q1* = D1

Q2* = D2

2

State or Transition Equations

Q0* = D0 = X . Y' . Q2'

Q1* = D1 = X . Q0

Q2* = D2 = Y' + Q1

3

Output Equations

Z1 = X . Q0 + Q1'

Z2= (Q1 . Q2)'

State Machine Analysis Example 2

4 **From the *state equations* and *output equation*, construct the *state transition/output table*:**

state name	Q2	Q1	Q0	XY = 00	XY = 01	XY = 11	XY = 10
A	0	0	0	100, 11	000, 11	000, 11	101, 11
B	0	0	1	100, 11	000, 11	010, 11	111, 11
C	0	1	0	100, 01	100, 01	100, 01	101, 01
D	0	1	1	100, 01	100, 01	110, 11	111, 11
E	1	0	0	100, 11	000, 11	000, 11	100, 11
F	1	0	1	100, 11	000, 11	010, 11	110, 11
G	1	1	0	100, 00	100, 00	100, 00	100, 00
H	1	1	1	100, 00	100, 00	110, 10	110,10

Q2* Q1* Q0*, Z1 Z2
(Next State, Outputs)

Transition Equations

Q0* = D0 = X . Y' . Q2'

Q1* = D1 = X . Q0

Q2* = D2 = Y' + Q1

Output Equations

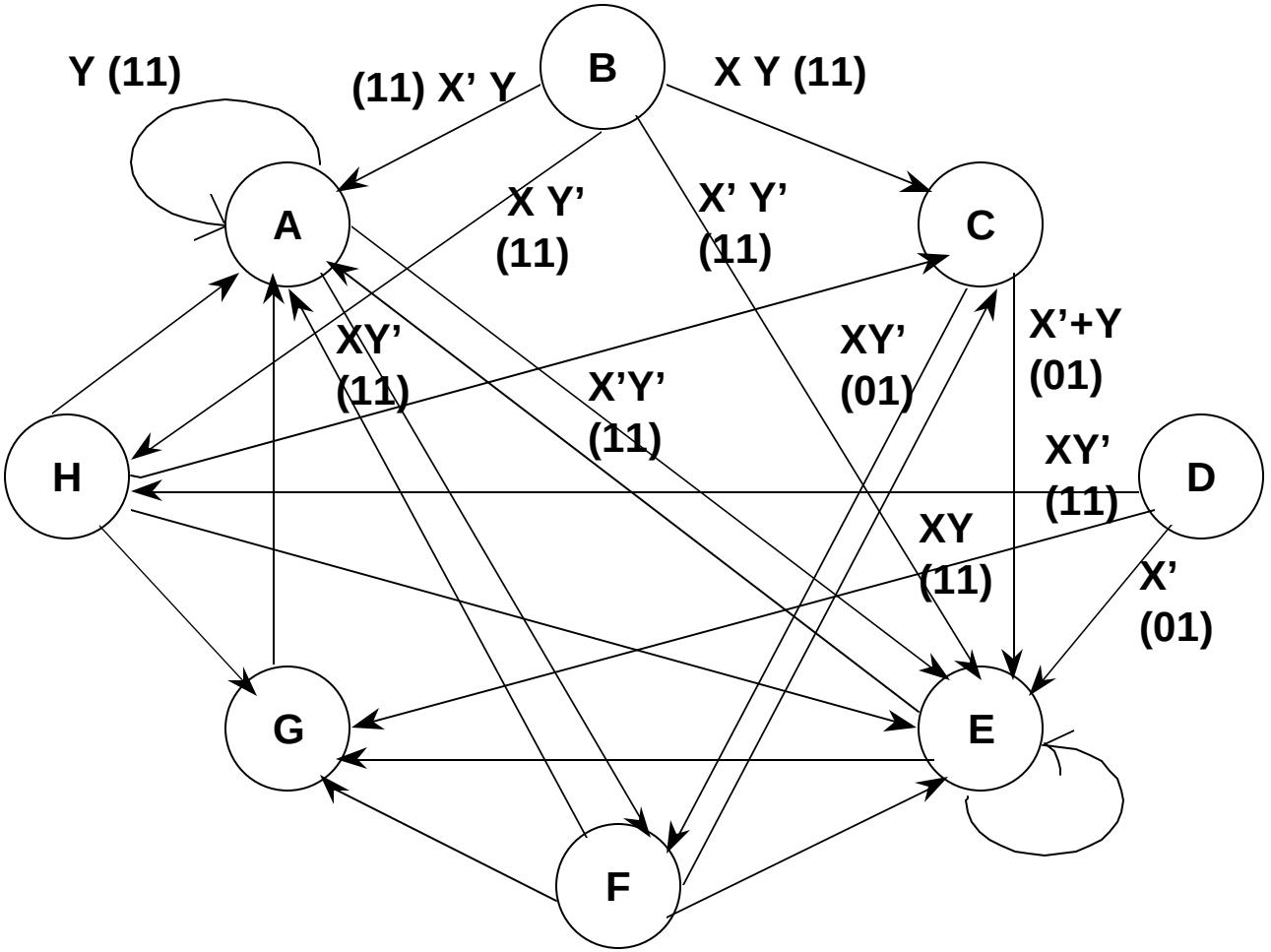
Z1 = X . Q0 + Q1'

Z2= (Q1 . Q2)'

State-machine Analysis Example 2: Transition/Output Table Using State Names

	XY			
S	00	01	11	10
A	E, 11	A, 11	A, 11	F, 11
B	E, 11	A, 11	C, 11	H, 11
C	E, 01	E, 01	E, 01	F, 01
D	E, 01	E, 01	G, 11	H, 11
E	E, 11	A, 11	A, 11	E, 11
F	E, 11	A, 11	C, 11	G, 11
G	E, 00	E, 00	E, 00	E, 00
H	E, 00	E, 00	G, 10	G,10
	S*, Z1 Z2			

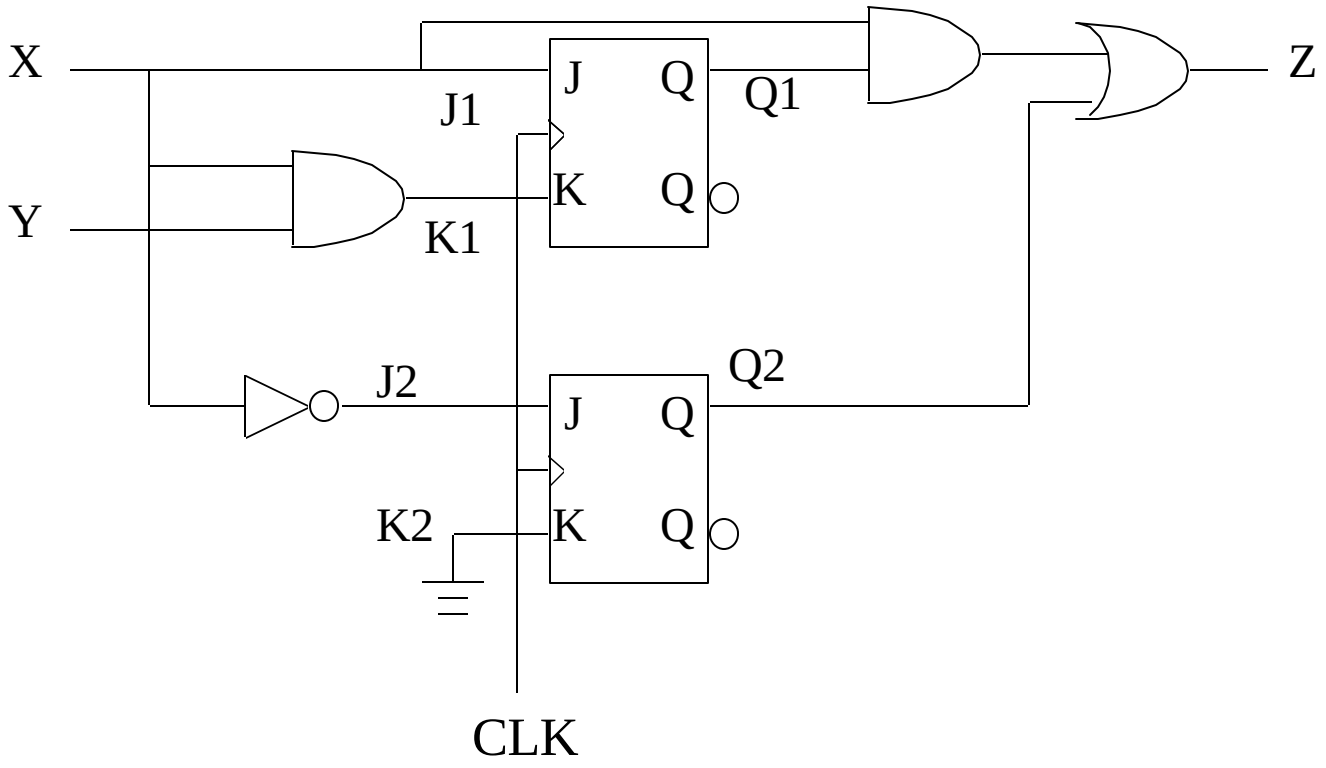
State-machine Analysis Example 2: State Diagram (incomplete)



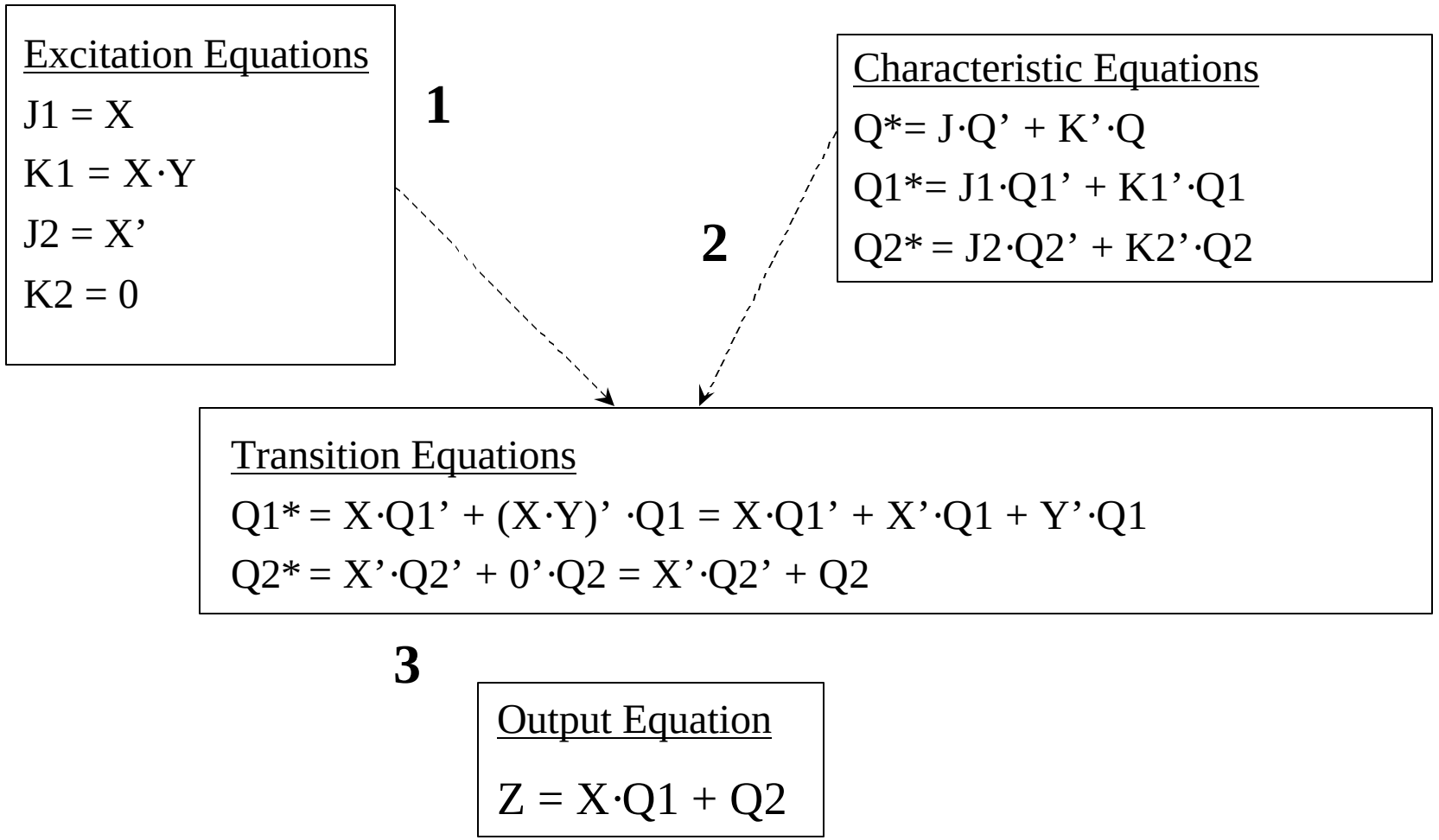
Arc: input expression (outputs) = expression (Z1 /Z2)

State Machine Analysis Example 3

Analyze the state machine:



State Machine Analysis Example 3



State Machine Analysis Example 3

4 **From the *state equations* and *output equation*, construct the *state transition/output table*:**

			XY			
S	Q1	Q2	00	01	11	10
A	0	0	01,0	01,0	10,0	10,0
B	0	1	01,1	01,1	11,1	11,1
C	1	0	11,0	11,0	00,1	10,1
D	1	1	11,1	11,1	01,1	11,1
			Q1* Q2*, Z			

Transition Equations

$Q1^* = X \cdot Q1' + X' \cdot Q1 + Y' \cdot Q1$

$Q2^* = X' \cdot Q2' + Q2$

Output Equation

$Z = X \cdot Q1 + Q2$

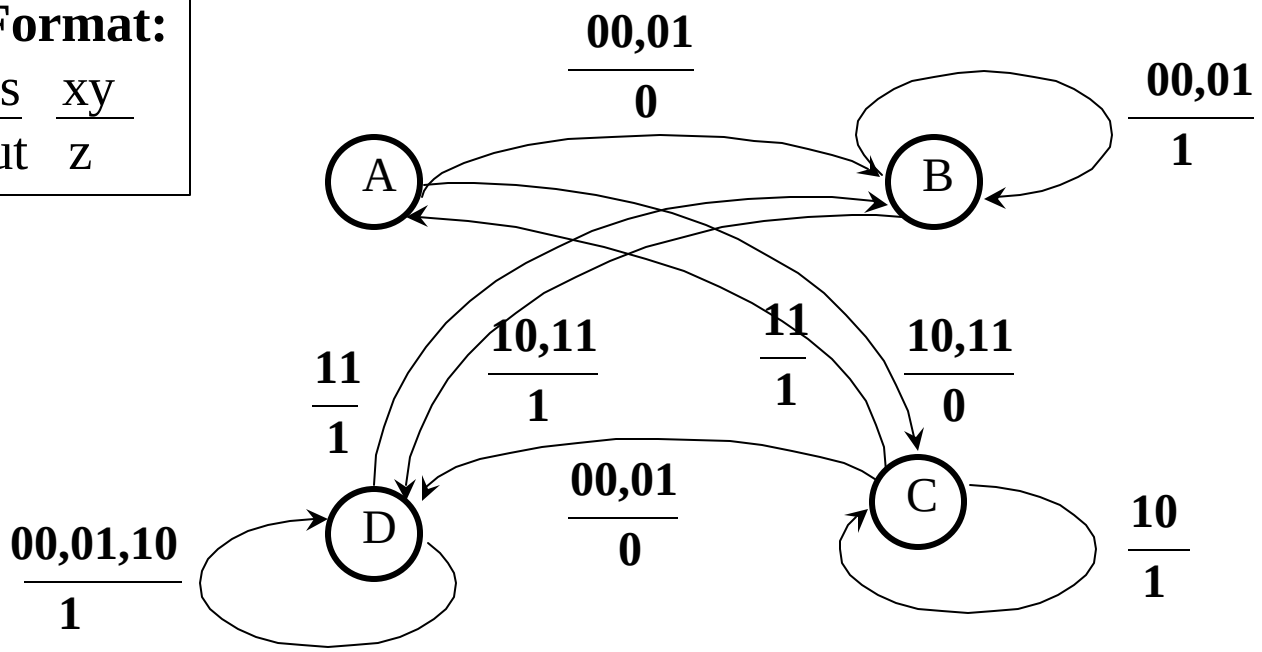
State-machine Analysis Example 3: Transition/Output Table Using State Names

S	XY 00	01	11	10
A	B,0	B,0	C,0	C,0
B	B,1	B,1	D,1	D,1
C	D,0	D,0	A,1	C,1
D	D,1	D,1	B,1	D,1

S*, Z

State-machine Analysis Example 3: State Diagram

Arc Format:
inputs xy
output z

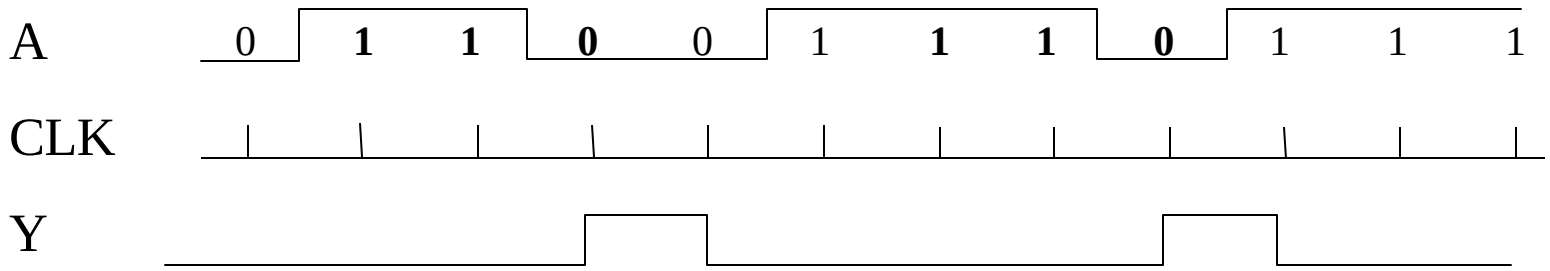


State Machine Design Procedure

1. Build state/output table (or state diagram) from word description using state names.
2. Minimize number of states (optional).
3. State Assignment: Choose state variables and assign bit combinations to named states.
4. Build transition/output table from state/output table (or state diagram) by substituting state variable combinations instead of state names.
5. Choose flip-flop type (D, J-K, etc.)
6. Build excitation table for flip-flop inputs from transition table.
7. Derive excitation equations from excitation table.
8. Derive output equations from transition/output table.
9. Draw logic diagram with excitation logic, output logic, and state memory elements.

State Machine Design Example 1: 110 Detector

- **Word description (110 input sequence detector):**
  - **Design a state machine with input A and output Y.**
  - **Y should be 1 whenever the sequence 1 1 0 has been detected on A on the last 3 consecutive rising clock edges (or ticks).**
  - **Otherwise, Y = 0**
  - **Note: this is a Moore machine, that is the output, Y, depends only on inputs at previous clocks rising edges , not on the current input.**
- **Timing diagram interpretation of word description (only rising clock edges are shown):**



State Machine Design Example 1: 110 Detector
Step1: Choosing States

- **Possible states (What does the state machine need to remember?):**
  - **Initial : power up, no clocks yet — Y = 0**
  - **No1s : first 1 not found — Y = 0**
  - **First1 : first 1 found — Y = 0**
  - **Two1s : at least 2 consecutive 1s found — Y = 0**
  - **ALL : found 1 1 0 — Y = 1**

- **Are all the states needed?**
  - **Notice: Initial is equivalent to NO1s**
  - **We can drop the state Initial and replace it with state No1s**

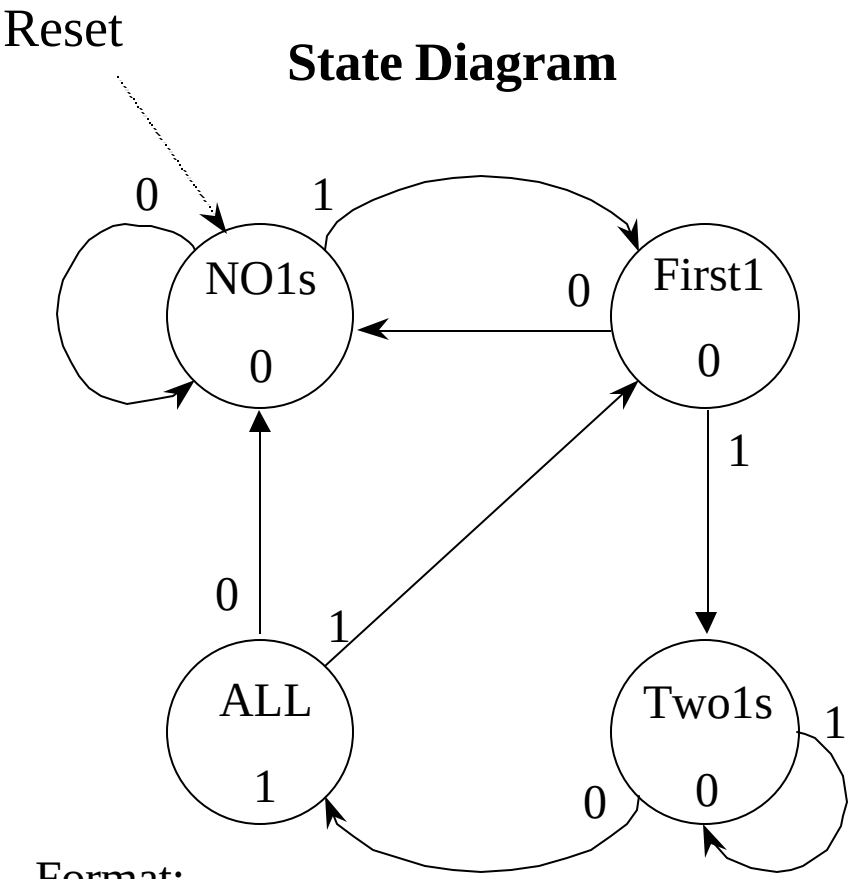
State Machine Design Example 1: 110 Detector

Step 1: State/Output Table and Diagram

State Table

S	A = 0	A = 1	Y
No1s	No1s	First1	0
First1	No1s	Two1s	0
Two1s	ALL	Two1s	0
ALL	No1s	First1	1
	S*		

State Diagram



Format:
Arc: input A
Node: state/output Y

Step3: State Assignment Considerations

- **Why does the choice of state assignment matter?**
  - Has a big effect on the complexity of excitation and output equations and thus on the amount of combinational logic needed.
- **How to find the best state assignment?**
  - The only known way is to try all assignments and determine the resulting equations.
    - **N = 2: $(2^2)! = 4! =$ 24 assignments for 2 state bits**
    - **N = 3: $(2^3)! = 8! =$ 40,320 assignments for three state bits.**
    - **N = 4: $(2^4)! = 16! =$ 20,922,789,888,000 assignments for 4 state bits!!!**

      THIS IS NOT PRACTICAL APPROACH!

  ∴ Use heuristic guidelines for pretty good assignments.

  This is still an active area of research!
- **There is no effective way to guarantee a "best" assignment. The heuristic methods sometimes perform poorly!**

State Assignment Strategies

- **Simplest Assignment:**
  - Straight binary, not best; purely arbitrary assignment.
- **One Hot Assignment:**
  - Redundant encoding, each flip-flop is assigned a state.
  - Uses the same number of bits as there are states (not useful in large designs).
  - Simple to assign; simple next state logic (no state decoding required)
  - Output logic is simple! One OR gate per Moore output!
- **Almost One Hot Assignment:**
  - Almost same as One Hot, but one less state bit.
  - Use all 0's to represent a state (usually INIT).
  - Must now decode state 0 if it is needed.
- **Decomposed Assignment:**
  - Use the "structure" of the state table to simplify next-state and output logic.
  - An "art" which requires much practice.

Example: State Assignment Strategies

Alternative Assignments

$Q_1..Q_4$	$Q_1..Q_5$	$Q_1Q_2Q_3$	$Q_1Q_2Q_3$
0000	00001	000	000
0001	00010	100	001
0010	00100	101	010
0100	01000	110	011
1000	10000	111	100
Almost One Hot	One Hot	Decomposed	Simplest

AB

S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1

– Example decomposition:

- **Initial State = all 0's for easy RESET**
- **INIT state is different, so use Q_1 = 1 for non-INIT states; thus** D1=1
- **Z = 1 in only 2 states, so use Q_2 =1 for states when Z = 1; thus** $Z = Q_2$
- **Use Q_3 = 1 for state transitions caused by A having the value of 1 (all destination states cause by A = 1, i.e. states A1 and OK1); thus** D3=A

THUS, simpler next state and output logic!

State Assignment Heuristic Guidelines

Starting from the highest priority to the lowest:

- **Choose initial coded state that's easy to produce at reset: (all 0's or 1's)**
  - **This simplifies the initialization circuitry.**
- **Freely use <u>any</u> of the 2^n state codes for best assignment**

  (i.e.. with s states, don't just use the first s integers 0,1,…,s-1)
- **Define specific bits or fields that have meaning with respect to input or output variables (decomposed codes).**
- **Consider using more than minimum number of state variables to allow for decomposed codes.**
- **Minimize number of state variables that change at each transition**
- **Simplify output logic.**

State Machine Design Example 1: 110 Detector

Step 3: State Assignment

- **Choose state variable assignments:**
  - Initial state all 0s
  - Q2 = last A, so Q2* = A
  - minimize number of transitions

			A		
Q1	Q2	S	0	1	Y
0	0	No1s	No1s	First1	0
0	1	First1	No1s	Two1s	0
1	1	Two1s	ALL	Two1s	0
1	0	ALL	No1s	First1	1
			S*		

State Machine Design Example 1: 110 Detector

Step 4: Transition/Output Table

- **Step 4: Build transition/output table from state/output table by substituting state variable combinations instead of state names.**

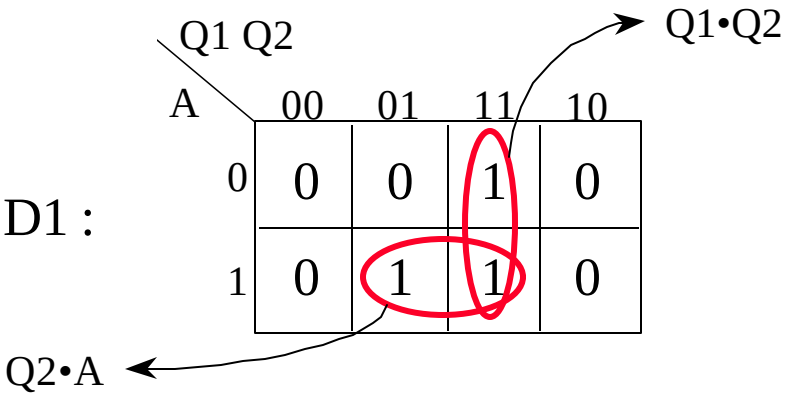
		A		
Q1	Q2	0	1	Y
0	0	00	01	0
0	1	00	11	0
1	1	10	11	0
1	0	00	01	1
		Q1* Q2*		
		=D1 D2	← Step 6	

- **Step 5: Choose D Flip-Flops , so Q*= D**
- **Step 6: Excitation table:**
  - **Same as Transition/output table with Q1*=D1, Q2*=D2**

State Machine Design Example 1: 110 Detector
Steps 7, 8 : Excitation/Output Equations

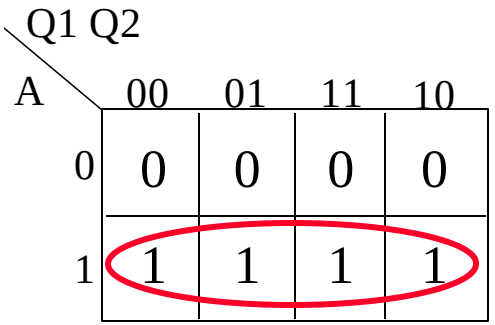
- **Step 7: Excitation equations: D1, D2 = F (A, Q1, Q2)**

D1 :



D1 = Q1•Q2 + Q2•A

D2 :



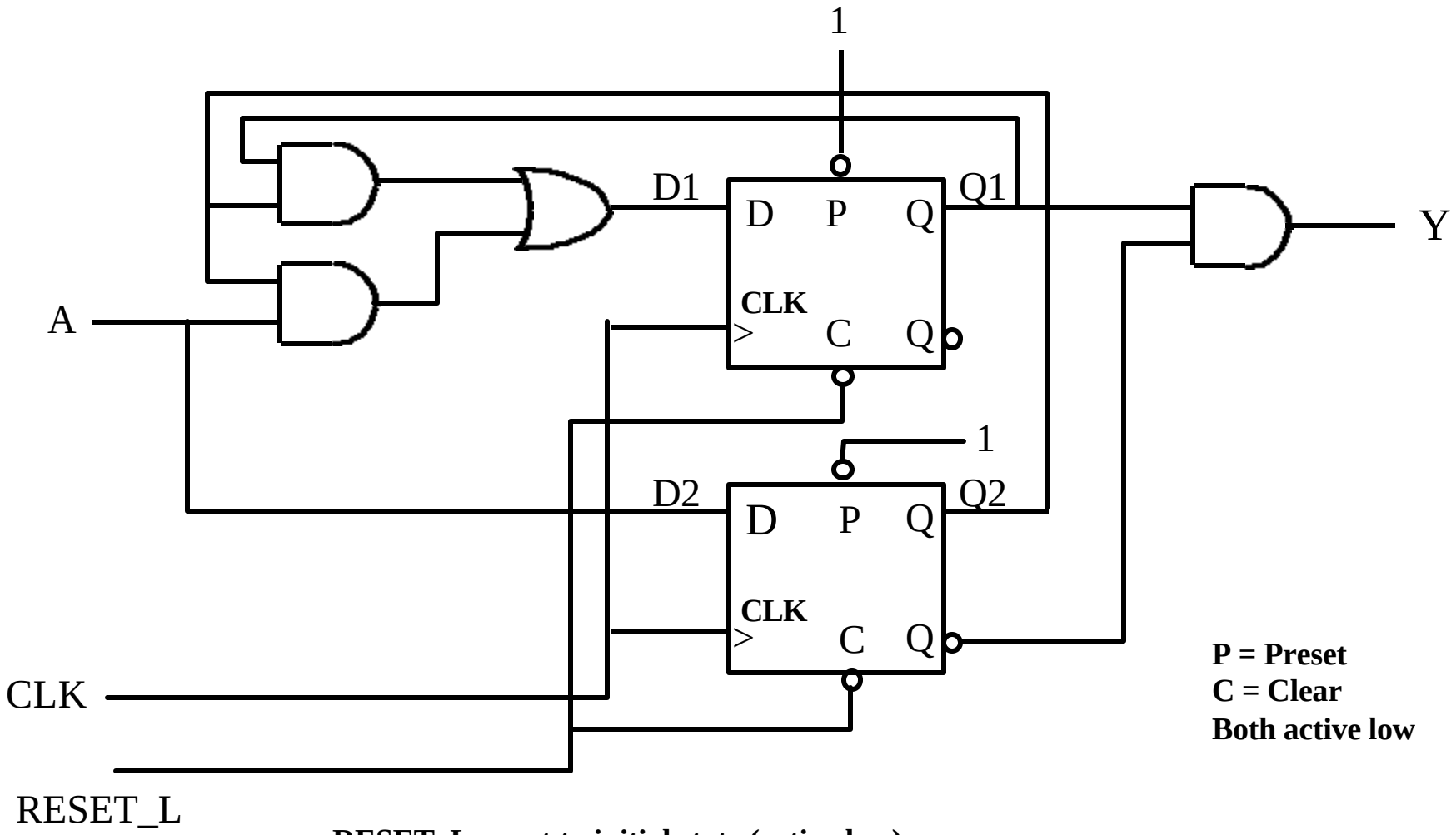
D2 = A (as planned!)

- **Step 8: Output equation: Y = G (Q1, Q2)**

  Y = Q1•Q2' (directly read from transition table)

State Machine Design Example 1: 110 Detector

Step 9: Logic Diagram

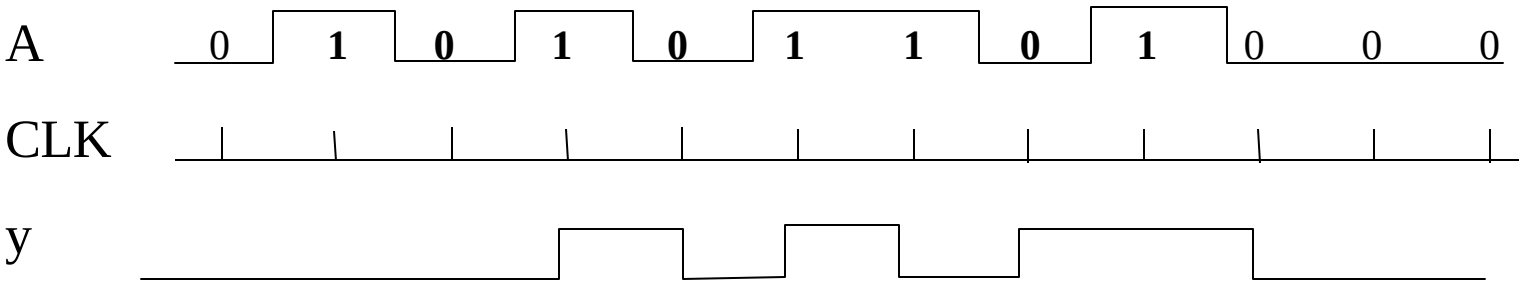


RESET_L reset to initial state (active low)

State Machine Design Example 2: 110/101 Detector

- **Word description (110/101 input sequence detector):**
  - **Design a state machine with input A and output Y.**
  - **Y = 1 when either sequence 1 1 0 <u>or</u> 1 0 1 has been detected on input A on the last 3 consecutive rising clock edges (or ticks).**
  - **Otherwise Y = 0**
  - **Note: Correct sequences <u>may overlap</u> and still be accepted.**

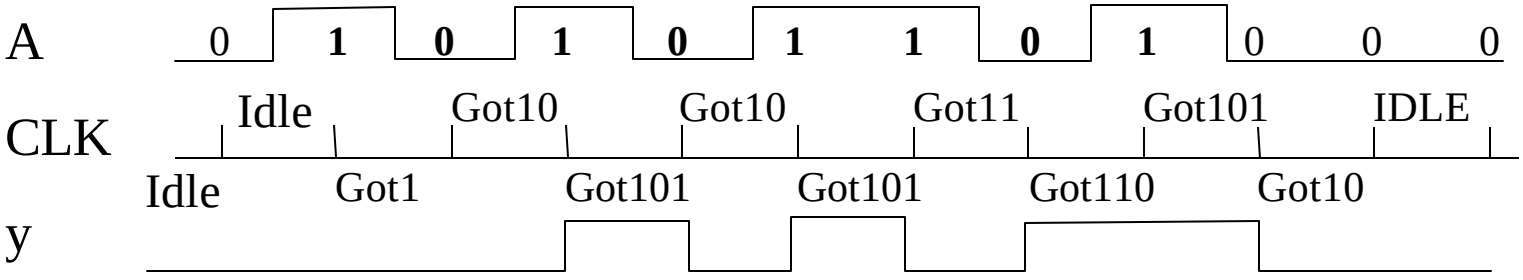
- **Timing diagram interpretation of word description (only rising clock edges are shown):**



State Machine Design Example 2: 110/101 Detector

Step1: Choosing States

- **Possible states (What does the state machine need to remember?):**
  - **Idle : Initial state, no starting 1 yet — Y = 0**
  - **Got1 : A = 1 on last tick — Y = 0**
  - **Got10 : Sequence A = 10 on last two ticks — Y = 0**
  - **Got101 : Sequence A = 101 on last three ticks — Y = 1**
  - **Got11 : Sequence A = 11 on last two ticks — Y = 0**
  - **Got110 : Sequence A = 110 on last three ticks — Y = 1**



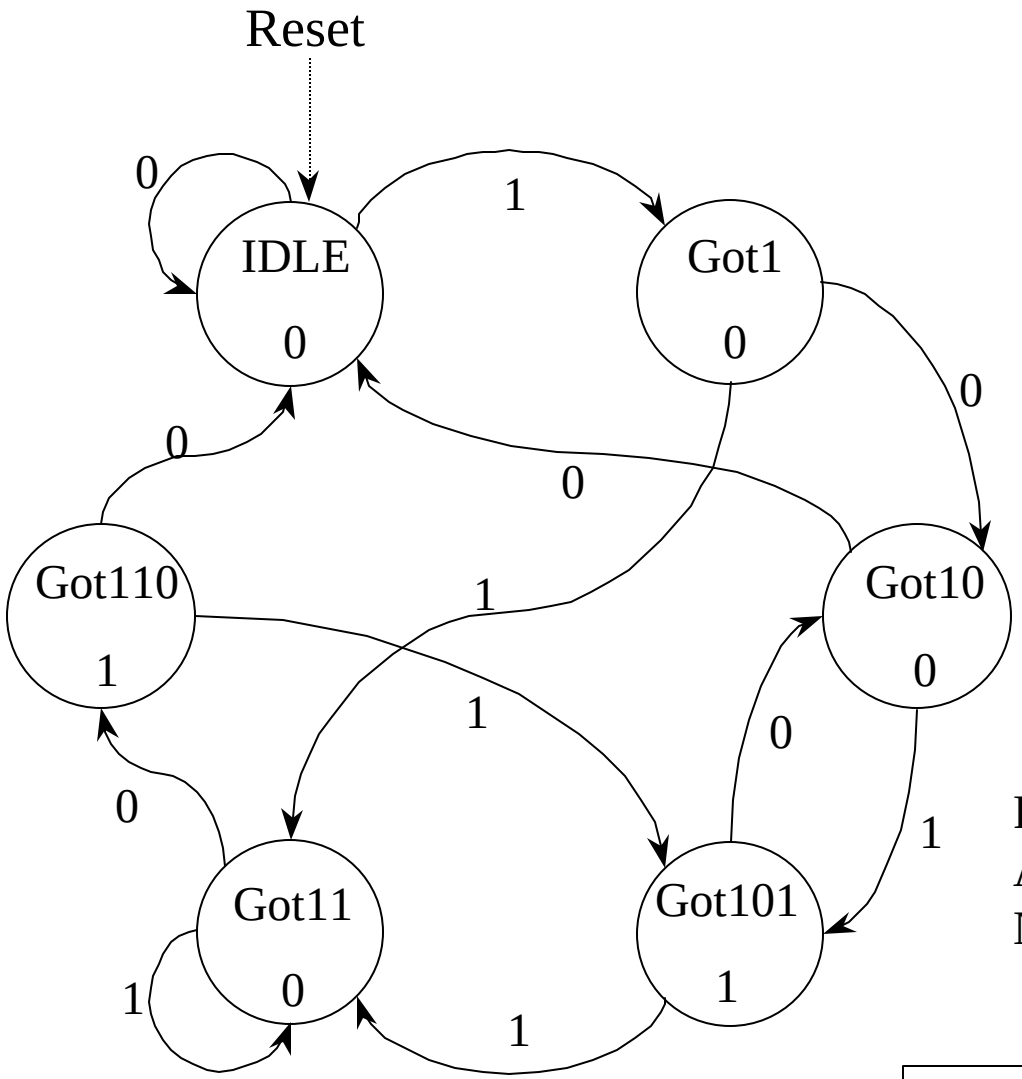
State Machine Design Example 2: 110/101 Detector

Step 1: State/Output Table

	A		
S	0	1	Y
IDLE	IDLE	Got1	0
Got1	Got10	Got11	0
Got10	IDLE	Got101	0
Got101	Got10	Got11	1
Got11	Got110	Got11	0
Got110	IDLE	Got101	1
	S*		

State Machine Design Example 2: 110/101 Detector

Step 1: State Diagram



Format:
Arc: input A
Node: state/output Y

State Machine Design Example 2: 110/101 Detector
Steps 3: State Assignment

- **Step 3: Choose state variable assignments :**
  - **Initial state all 0s**
  - **Q1 = Y**
  - **Q3 = last A, so Q3* = A**
  - **minimum number of transitions**

From Step 1:

Q1 Q2 Q3	S	A = 0	A = 1	Y
0 0 0	IDLE	IDLE	Got1	0
0 0 1	Got1	Got10	Got11	0
0 1 0	Got10	IDLE	Got101	0
1 1 1	Got101	Got10	Got11	1
0 1 1	Got11	Got110	Got11	0
1 1 0	Got110	IDLE	Got101	1
		S*		

State Machine Design Example 2: 110/101 Detector

- **Step 4: Transition/output table**
- **Step 5: Choose D Flip-flops**
- **Step 6: Excitation table**
  - **Same as Transition table**

Q1 Q2 Q3	A = 0	A = 1	Y
0 0 0	000	001	0
0 0 1	010	011	0
0 1 0	000	111	0
1 1 1	010	011	1
0 1 1	110	011	0
1 1 0	000	111	1
Unused states? 1 0 0	ddd	ddd	d
1 0 1	ddd	ddd	d

~~Q1*Q2* Q3*~~

=D1 D2 D3

State Machine Design Example 2: 110/101 Detector

Steps 7: Excitation Equations

- **Step 7: Excitation equations**
  - D1, D2, D3 = F (A, Q1, Q2, Q3)

D1 = Q1'•Q2•Q3•A' + Q2•Q3'•A

D2 = Q2•A + Q3

D3 = A (as planned!)

D1 :

Q3 A \ Q1 Q2	00	01	11	10
00				d
01		1	1	d
11				d
10		1		d

D2 :

Q3 A \ Q1 Q2	00	01	11	10
00				d
01		1	1	d
11	1	1	1	d
10	1	1	1	d

D3 :

Q3 A \ Q1 Q2	00	01	11	10
00				d
01	1	1	1	d
11	1	1	1	d
10				d

State Machine Design Example 2: 110/101 Detector

Step 8: Output Equations

- **Step 8: Output equation**
  - Y = Q1 (as planned!)

- **Step 9: Logic diagram**
  - (3) D-Flip-flops + (3) 2-input gates + (1) 3-input AND gate + (1) 4-input AND gate
  - Draw the diagram.

D1 = Q1'•Q2•Q3•A' + Q2•Q3'•A

D2 = Q2•A + Q3

D3 = A

State Machine Design Using J-K Flip-Flops

- **State machine design step 6 (building excitation table for flip-flop inputs from transition table):**
  - **When using D flip-flops, since the next state Q* = D, the excitation table is the same as the transition table with Q* replaced with D.**
  - **In the case of J-K flip-flops, the next state is given by:**
    Q* = J . Q' + K'. Q
  - **In this case we cannot rearrange the characteristic equation to find separate equations for J, K.**
  - **Instead an application (or excitation) table for J-K flip-flops is used to obtain the corresponding values of J, K for a given Q to Q* transition:**

Q	Q*	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

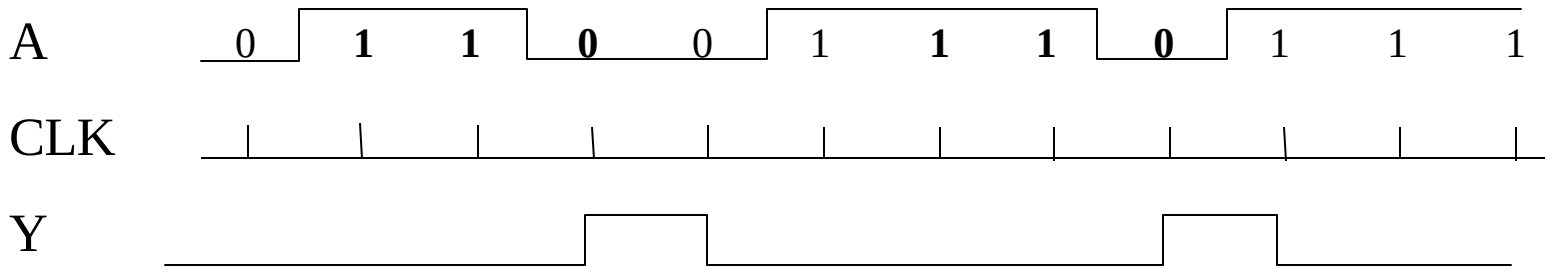
J-K Flip-Flop Excitation Table

State Machine Design Example 1:

110 Detector (Repeated Using J-K Flip-Flops)

- **Word description (110 input sequence detector):**
  - **Design a state machine with input A and output Y.**
  - **Y should be 1 whenever the sequence 1 1 0 has been detected on A on the last 3 consecutive rising clock edges (or ticks).**
  - **Otherwise, Y = 0**

- **Timing diagram interpretation of word description (only rising clock edges are shown):**



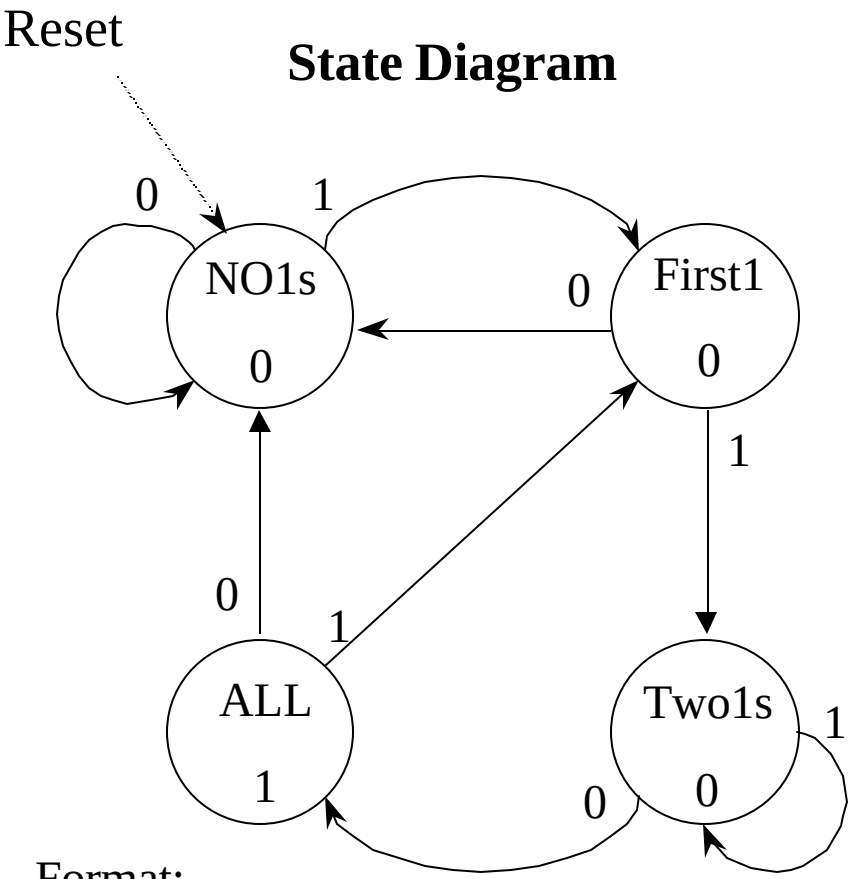
State Machine Design Example 1: 110 Detector

Step 1: State/Output Table and Diagram

State Table

S	A = 0	A = 1	Y
No1s	No1s	First1	0
First1	No1s	Two1s	0
Two1s	ALL	Two1s	0
ALL	No1s	First1	1
	S*		

State Diagram



Format:
Arc: input A
Node: state/output Y

State Machine Design Example 1: 110 Detector Using J-K Flip-flops

- **Steps 1-4: No change.**

Transition Table (step 4):

Q1	Q2	A = 0	A = 1	Y
0	0	00	01	0
0	1	00	11	0
1	1	10	11	0
1	0	00	01	1
		Q1* Q2*		

Q	Q*	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

J-K Flip-Flop Excitation Table

Excitation table (Step 6):

Q1	Q2	A = 0	A = 1	Y
0	0	0d, 0d	0d, 1d	0
0	1	0d, d1	1d, d0	0
1	1	d0, d1	d0, d0	0
1	0	d1, 0d	d1, 1d	1
		J1 K1, J2 K2		

- **Step 5: Choose J-K Flip-Flops**
- **Step 6: Excitation table: Use J-K Flip-Flop Excitation Table.**

State Machine Design Example 1: 110 Detector Using J-K FF
Steps 7, 8 : Excitation/Output Equations

- **Step 7: Excitation equations: J1, K1, J2, K2 = F (A, Q1, Q2)**

J1 :

A \ Q1 Q2	00	01	11	10
0	0	0	d	d
1	0	1	d	d

J1 = Q2•A

J2 :

A \ Q1 Q2	00	01	11	10
0	0	d	d	0
1	1	d	d	1

J2 = A

K1 :

A \ Q1 Q2	00	01	11	10
0	d	d	0	1
1	d	d	0	1

K1 = Q2'

K2 :

A \ Q1 Q2	00	01	11	10
0	d	1	1	d
1	d	0	0	d

K2 = A'

- **Step 8: Output equation: Y = G (Q1, Q2)**

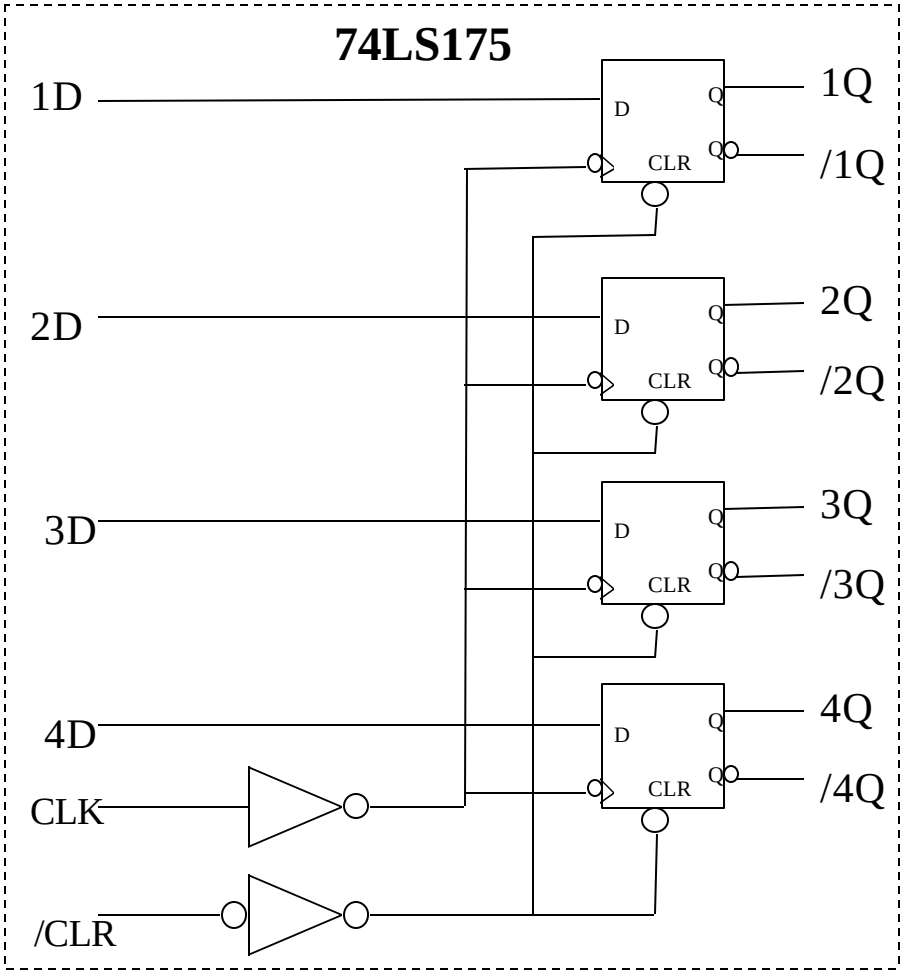
  Y = Q1•Q2' (directly read from transition table)

Registers & Counters

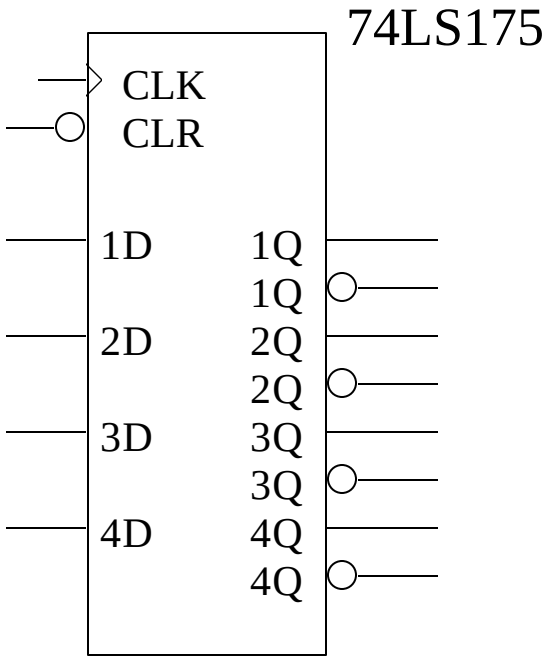
- **Registers.**
- **Shift Registers:**
  - **Serial in, serial out shift register**
  - **Serial in, parallel out shift register**
  - **Parallel in, serial out shift register**
  - **Parallel in, parallel out shift register**
  - **Shift Register Applications**
- **Counters:**
  - **Ripple Counters**
  - **Synchronous Counters**
  - **Counter Applications**

Registers

- **An n-bit register is a collection of n D flip-flops with a common clock used to store n related bits.**

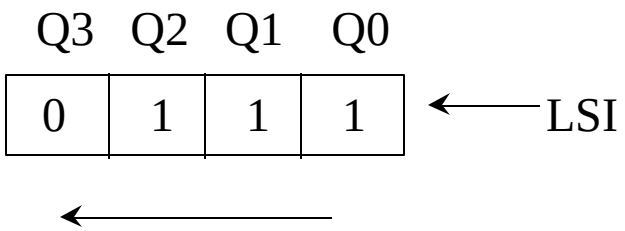


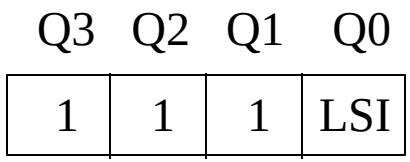
Example: 74LS175 4-bit register



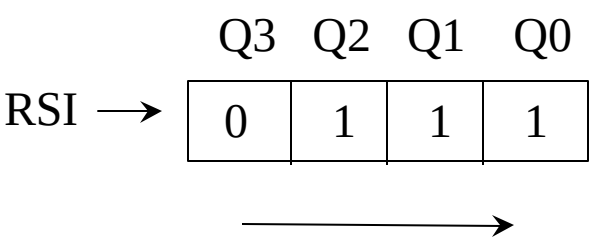
Shift Registers

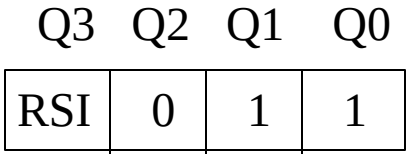
- **Multi-bit register that moves stored data bits left/right (1 bit position per clock cycle)**
  - **Shift Left is towards MSB**



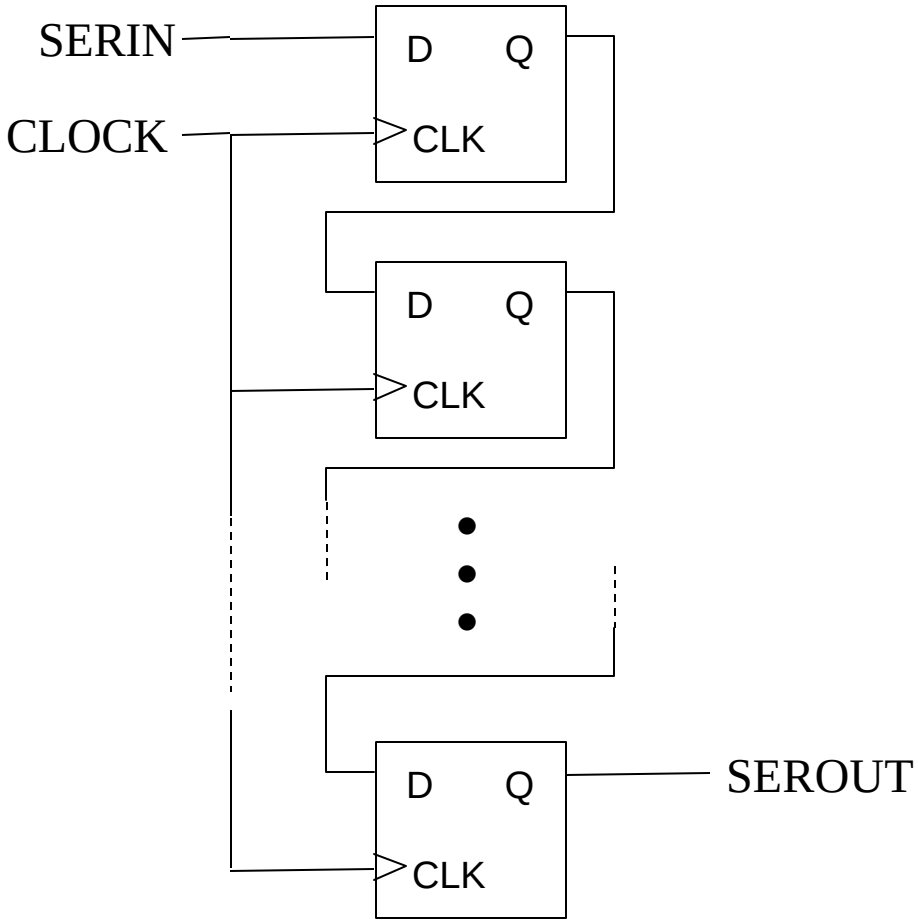


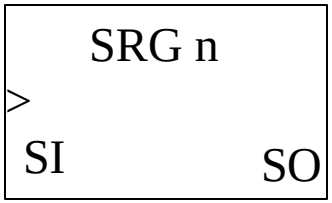
  - **Shift Right (or Shift Up) is towards MSB**





Serial In, Serial Out Shift Register





For a n-bit SRG:
Serial Out = Serial In delayed by n clock period

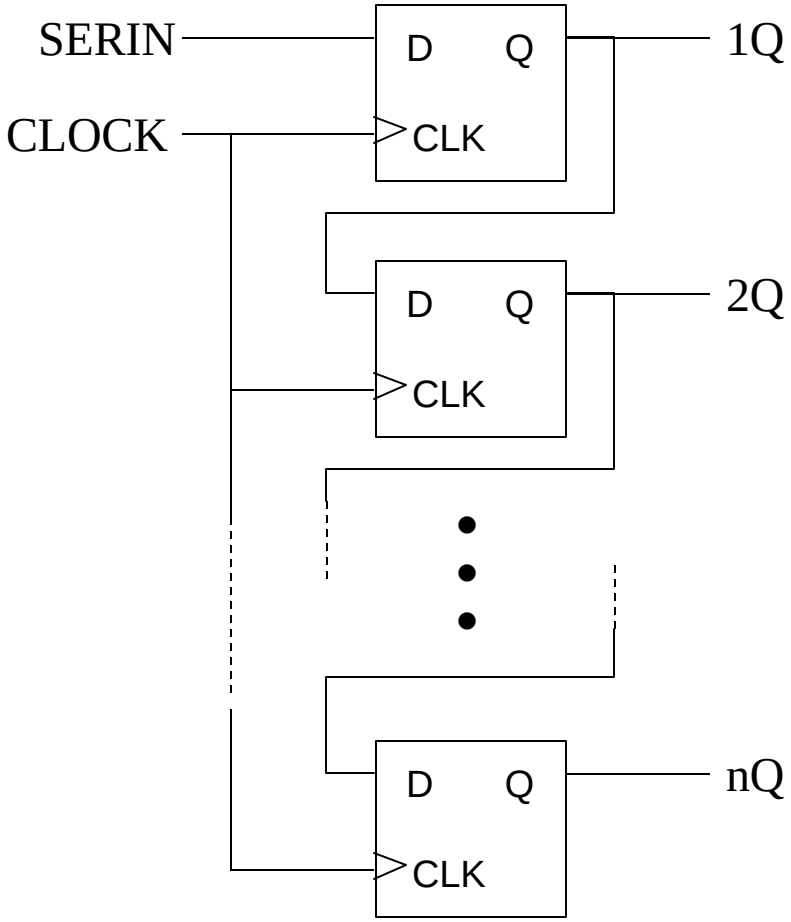
4-bit shift register example:

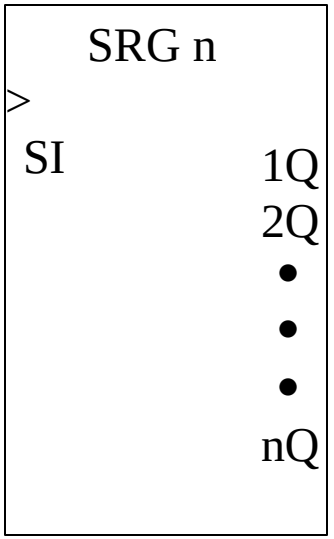
serin: 1 0 1 1 0 0 1 1 1 0

serout: - - - - 1 0 1 1 0 0

clock:

Serial In, Parallel Out Shift register





(SO)

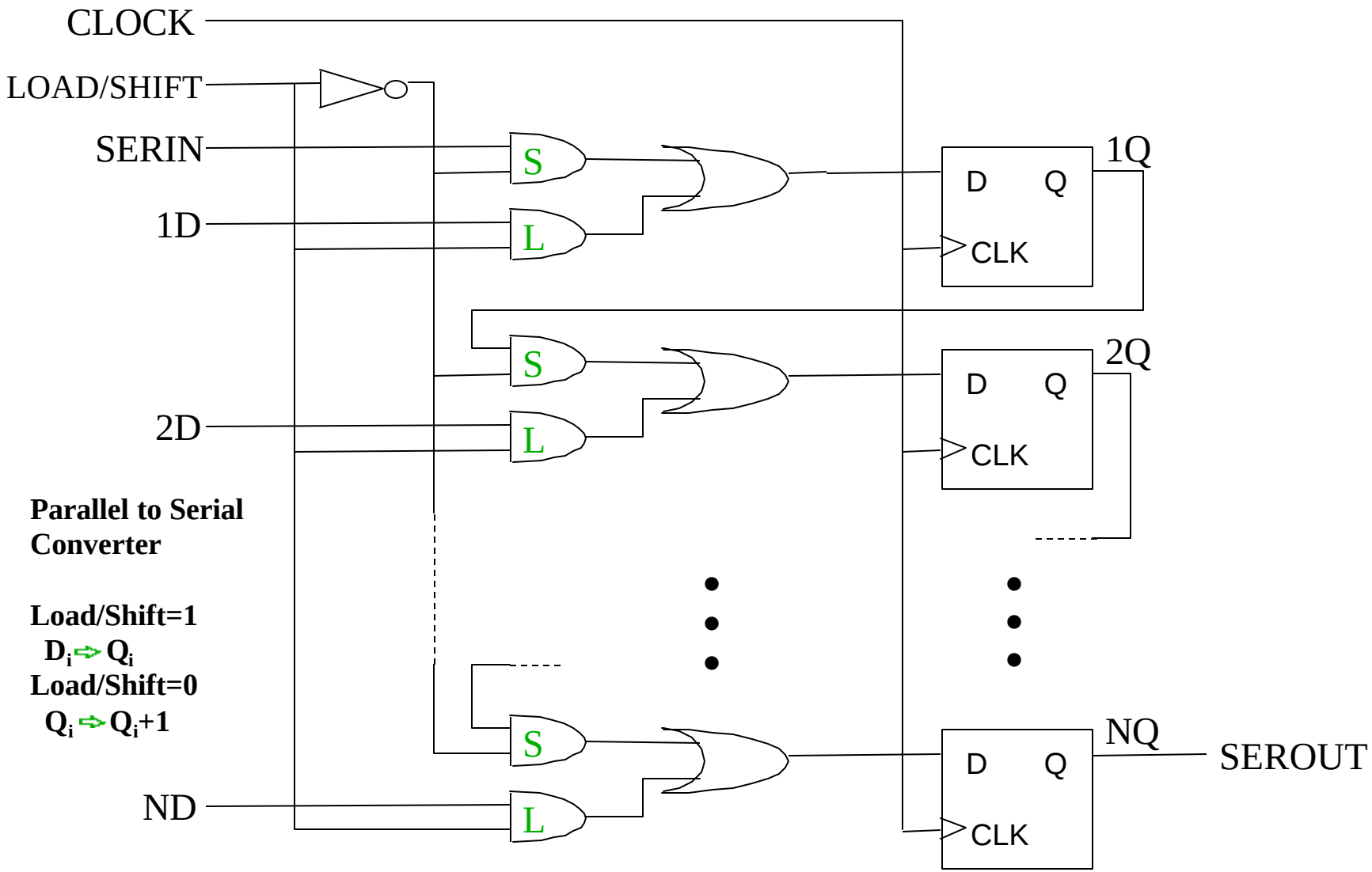
Serial to Parallel Converter

4-bit shift register example:

serin:	1 0 1 1 0 0 1 1 1 0
1Q:	- 1 0 1 1 0 0 1 1 1
2Q:	- - 1 0 1 1 0 0 1 1
3Q:	- - - 1 0 1 1 0 0 1
4Q:	- - - - 1 0 1 1 0 0

clock:

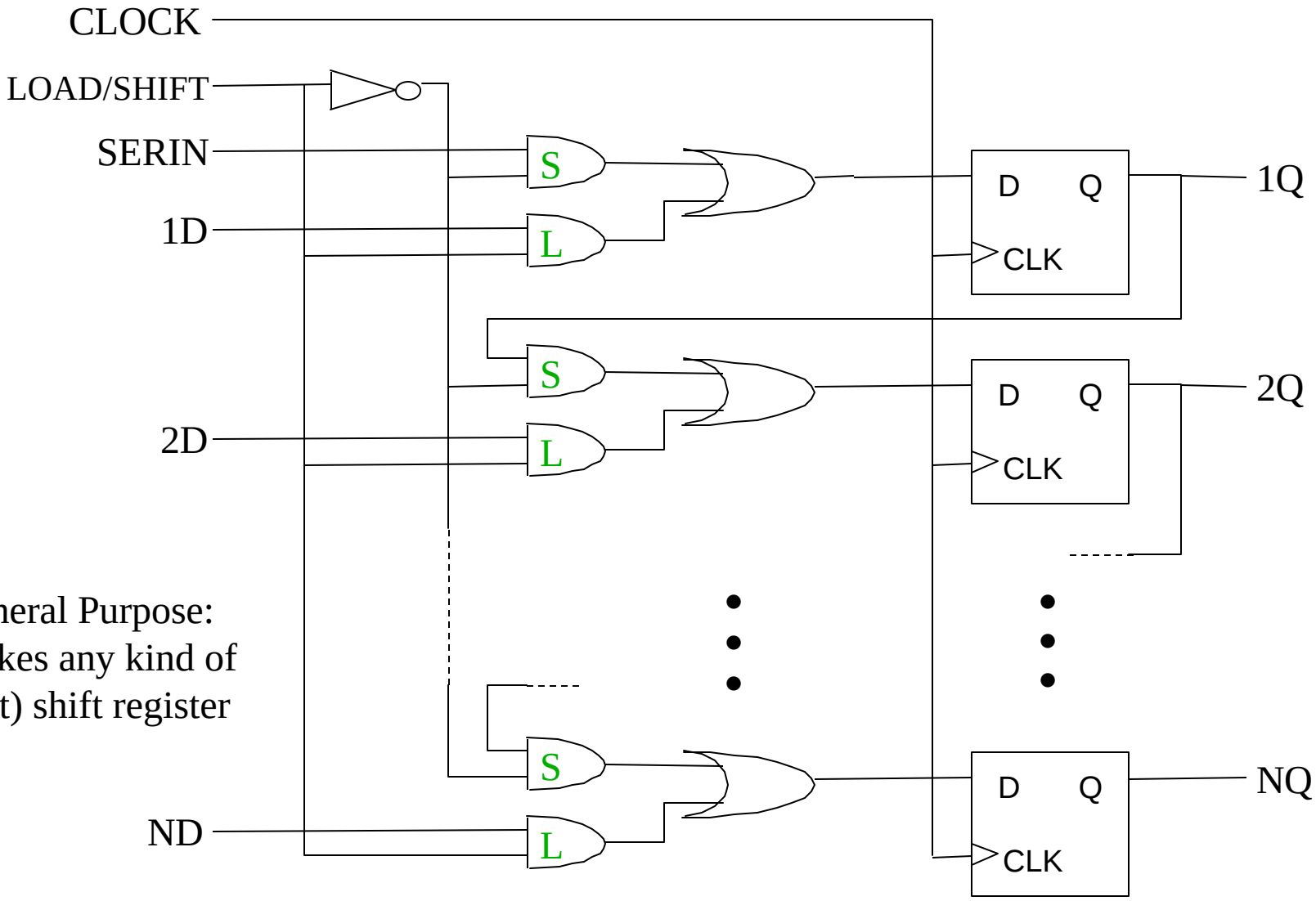
Parallel In, Serial Out Shift Register



Parallel to Serial Converter

Load/Shift=1
$D_i \Rightarrow Q_i$
Load/Shift=0
$Q_i \Rightarrow Q_i+1$

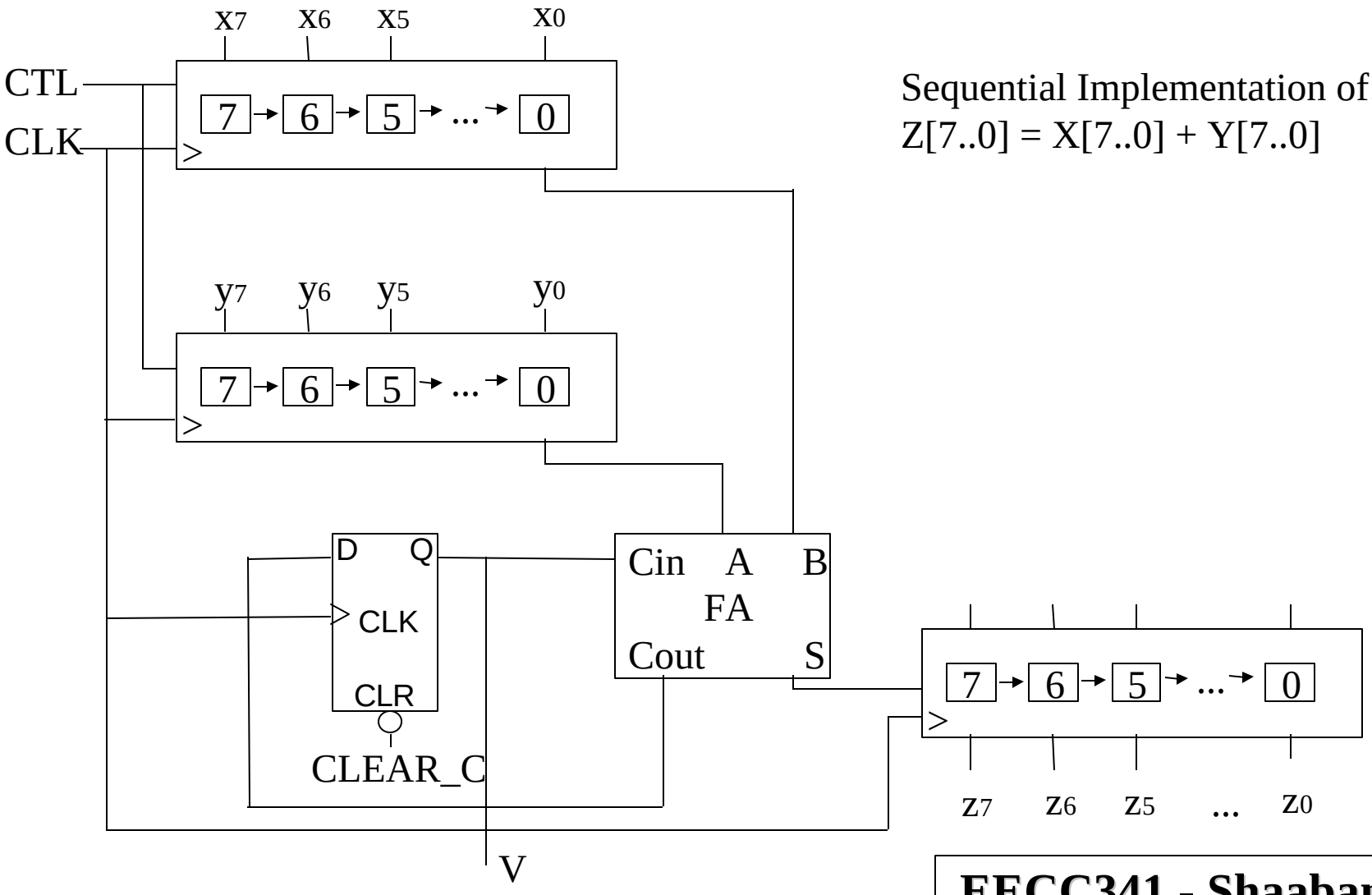
Parallel In, Parallel Out Shift Register



General Purpose:
Makes any kind of
(left) shift register

Shift Register Applications Example:

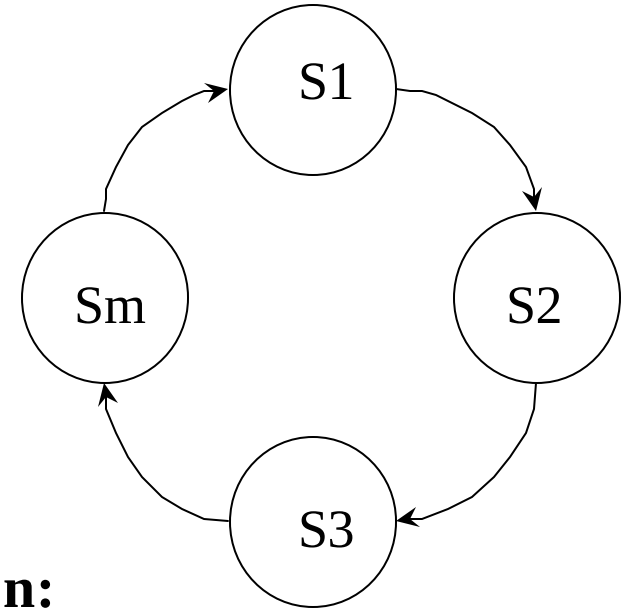
8-Bit Serial Adder



Sequential Implementation of:
Z[7..0] = X[7..0] + Y[7..0]

Counters

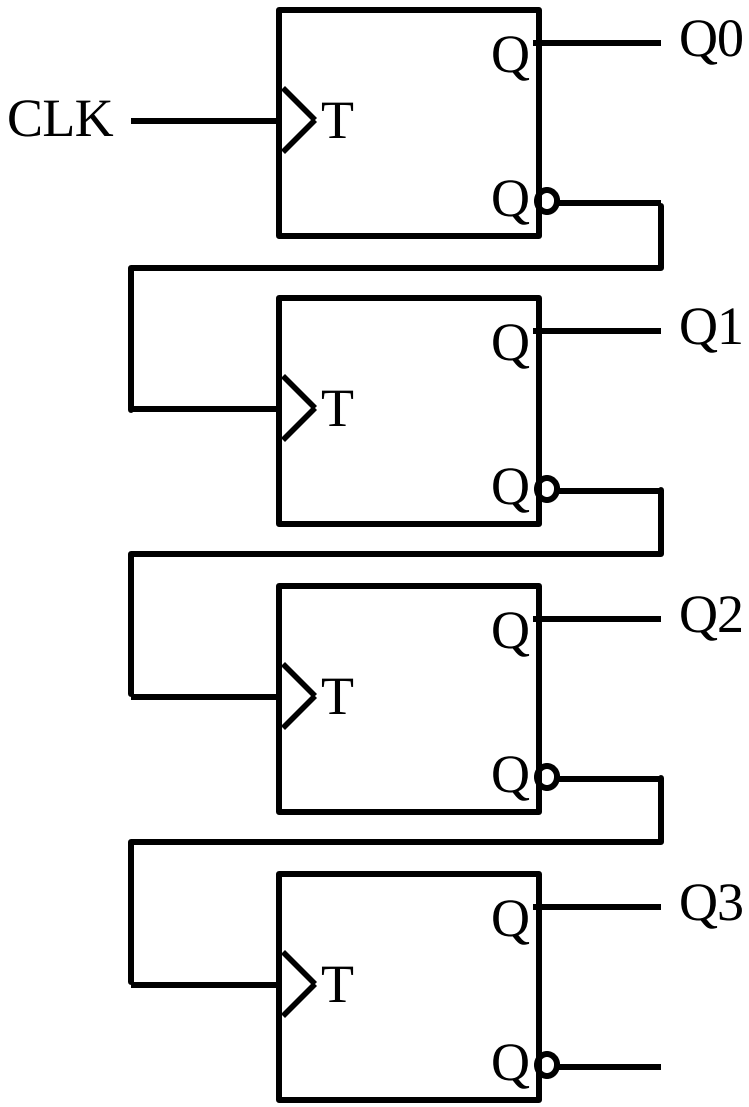
- **Clocked sequential circuit with single-cycle state diagram**
  - **Modulo-m counter = divide-by-m counter**



  - **Most Common:**

n-bit binary counter, where $m = 2^n$ ➔ n flip-flops, counts 0 … 2^n-1

4-bit Ripple Counter



1 bit

divide-by-2

2 bit

divide-by-4

3 bit

divide-by-8

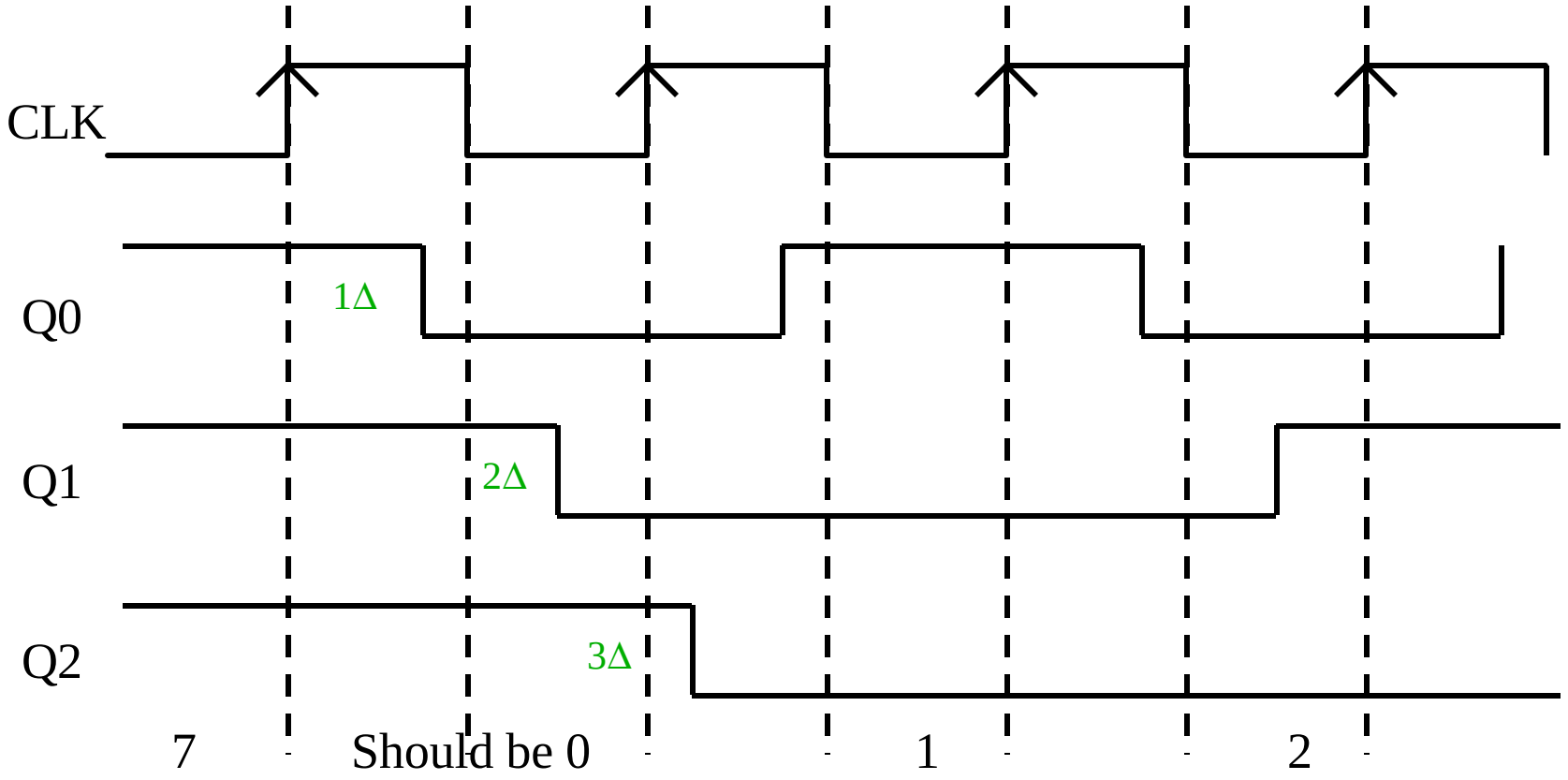
4 bit

divide-by-16

Uses Minimal Logic

Ripple Counter Problem

$n \bullet T_{CQ}$ for MSB change for n-bit ripple counter => minimum clk period

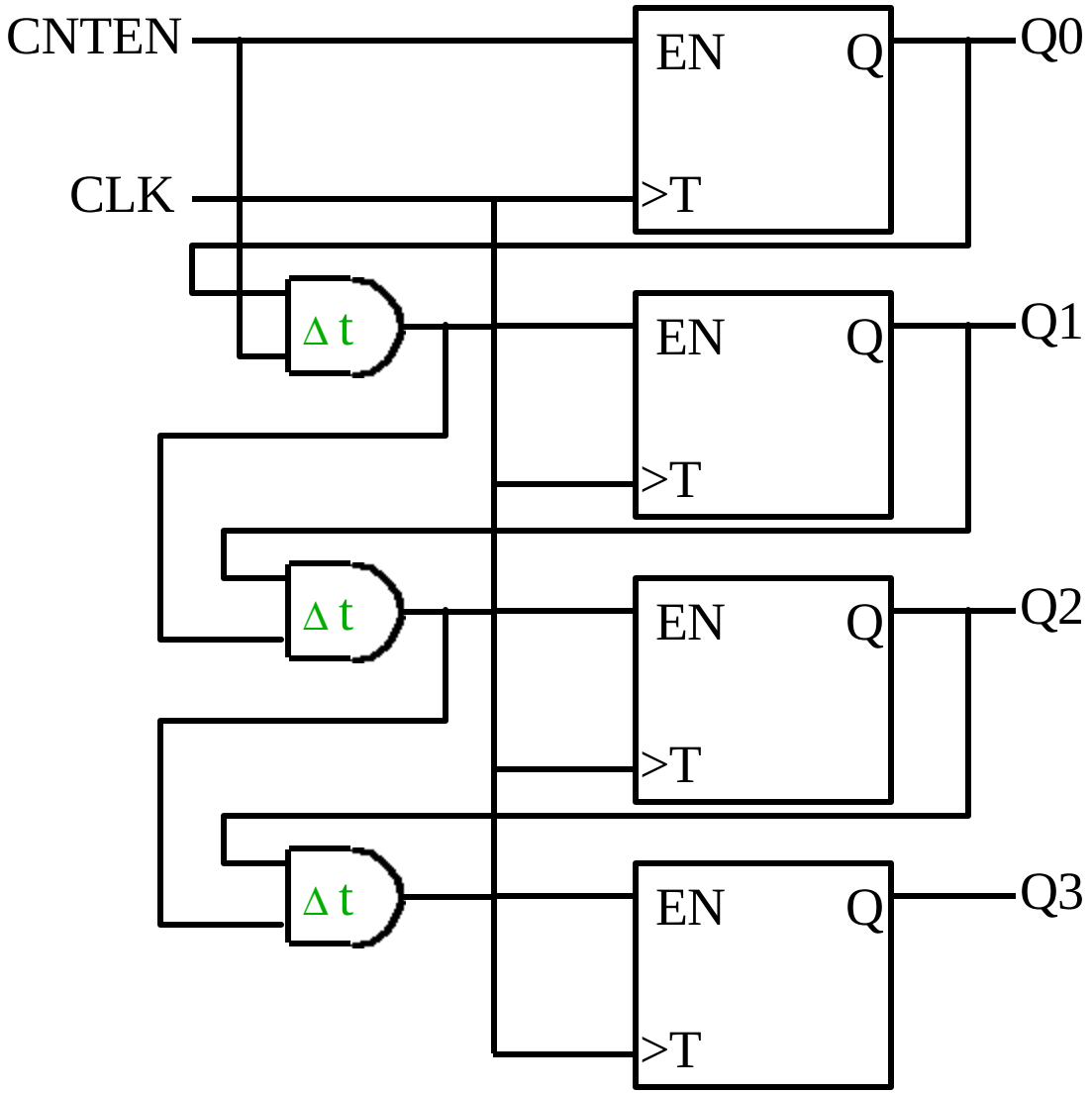


Synchronous Counters

- All clock inputs connected to common CLK signal
  - All flip-flop outputs change simultaneously t_{CQ} after CLK
  - Faster than ripple counters
  - More complex logic
  - Most frequently used type of counter

Synchronous Serial Counter

- **Flip-flops enabled when all lower flip-flops = 1.**
- **Enable propagates serially — limits speed**
- **Requires $(n-1) \Delta t < T_{CLK}$**
- **All outputs change simultaneously t_{CQ} after CLK↑**



Synchronous Parallel Counter

- **Single-level enable logic per flip-flop**
- **Fastest and most complex type of counter**
- **Requires D t < T_{CLK}**
- **All outputs change simultaneously t_{CQ} after CLK↑**