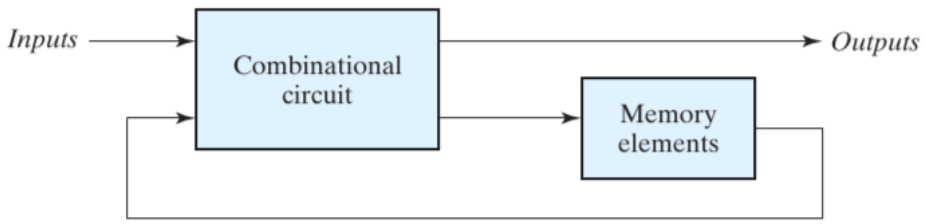
Haricharan

Digital Systems

May 23

Sequential circuits



Specified by a time sequence of inputs, outputs, internal states

- Synchronous →
  - happens at a particular clock frequency
  - action only at discrete instance
  - easy to define specific states
- Asynchronous →
  - if we know the time it takes to propagate through a gate or conducting line
  - propagates delay of logic gates
  - may become unstable

Synchronous sequential circuit

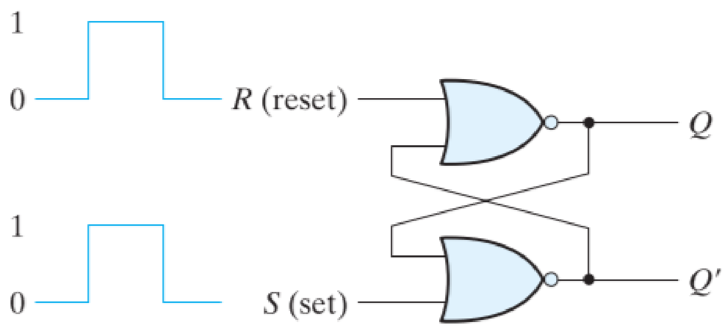
Latch: storage element changes state based on input level
Flip-flop: storage element changes state based on input transition (clock transition)

Latches

1. SR(set-reset) Latch
NOR implementation: Inputs zero by default, become 1 when triggered

S	R	Q	Q'
1	0	**1**	0 → *Set condition*
0	0	1	0 after S = 1, R = 0
0	1	**0**	1 → *Reset condition*

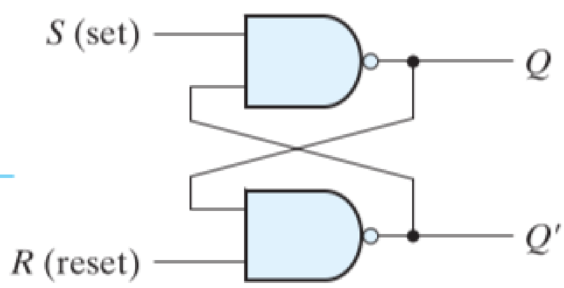
S	R	Q	Q'
0	0	0	1, after S = 0, R = 1
1	1	0	0 → *Forbidden state*



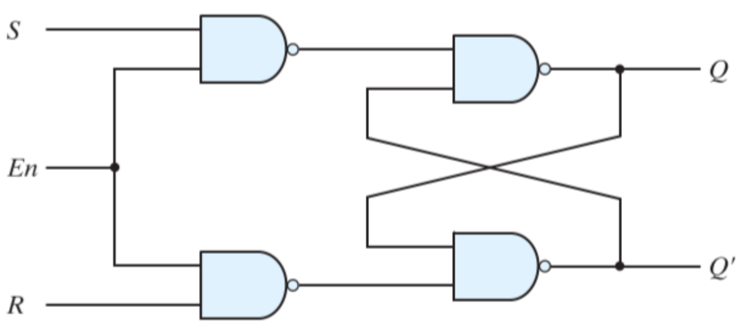
(a) Logic diagram

NAND implementation: Inputs 1 by default, zero when triggered

S	R	Q	Q'
1	0	**0**	1 → *Set condition*
1	1	0	1 after S = 1, R = 0
0	1	**1**	0 → *Reset condition*
1	1	1	0, after S = 0, R = 1
0	0	1	1 → *Forbidden state*



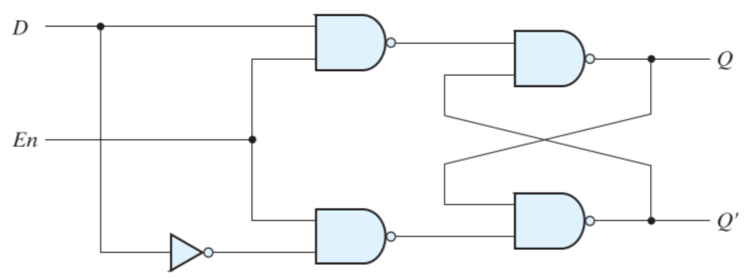
NAND implementation with enable pin:



E	S	R	Q	Q'

E	S	R	Q	Q'
0	X	X	No change	No change
1	1	1	No change	No change
1	0	1	1	0
1	1	0	0	1
1	0	0	1	1

2. D Latch (Transparent latch)

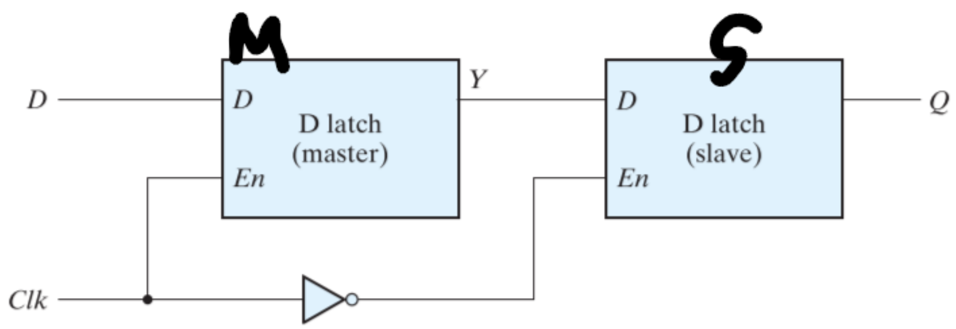


(a) Logic diagram

E	D	Q	Q'
0	X	No change	No change
1	0	0	1
1	1	1	0

Flip-flop

Negative edge triggered master-slave D Flip-flop

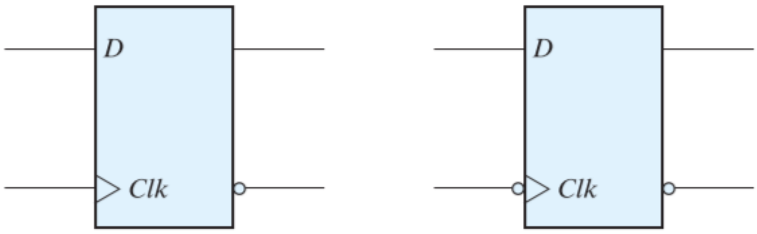


Clk	Y	Q
0	No change	Y
1	D	No change
0 → 1	No change → D	Y → No change
1 → 0	D → No change	No change → Y = D

> **Important**
>
> **Q(t + 1) = D**

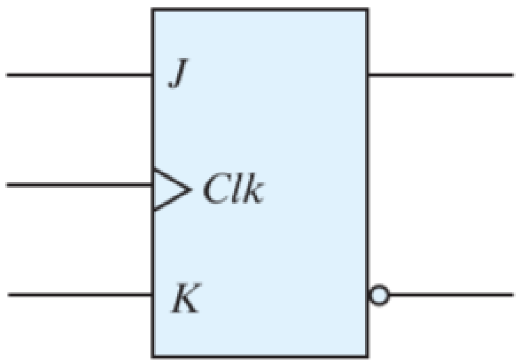
> It is example of Moore machine, output is independent of previous state

Positive triggered D flip-flop:
Complement clock



(a) Positive-edge (a) Negative-edge

JK Flip flop



D = JQ' + K'Q

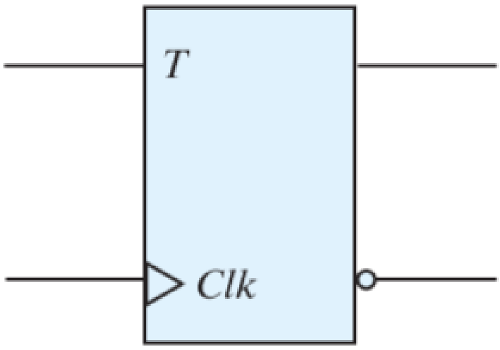
J	K	D	Q (t + 1)
0	0	Q	Q
0	1	0	0
1	0	1	1
1	1	1	Q'

Excitation Tables

A_t	A_{t+1}	J_A/J_B	K_A/J_B

A_t	A_{t+1}	J_A / J_B	K_A / J_B
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

T flip-flop
set J = K in JK flip-flop



(c) Graphic symbol

T	Q(t + 1)
0	Q(t)
1	Q'(t)

$Q(t+1) = T \oplus Q$

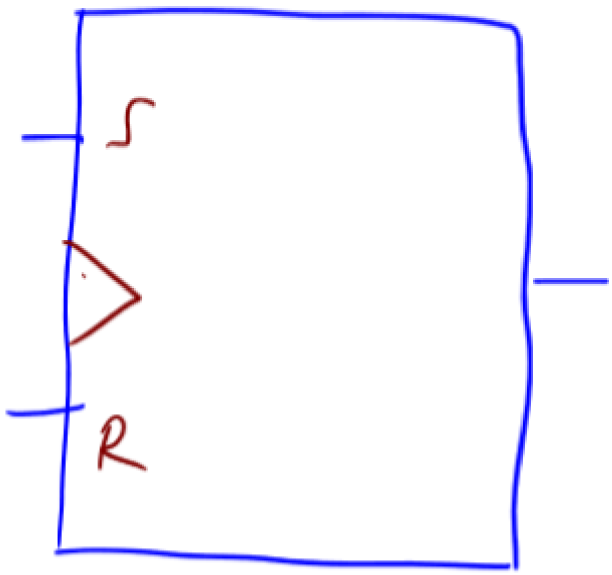
Excitation Table:

A_t	A_{t+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

SR Flip-flop

S	R	Q	Q'
0	0	No change	No change

S	R	Q	Q'
0	1	0	1
1	0	1	0
1	1	1	1



Excitation Table:

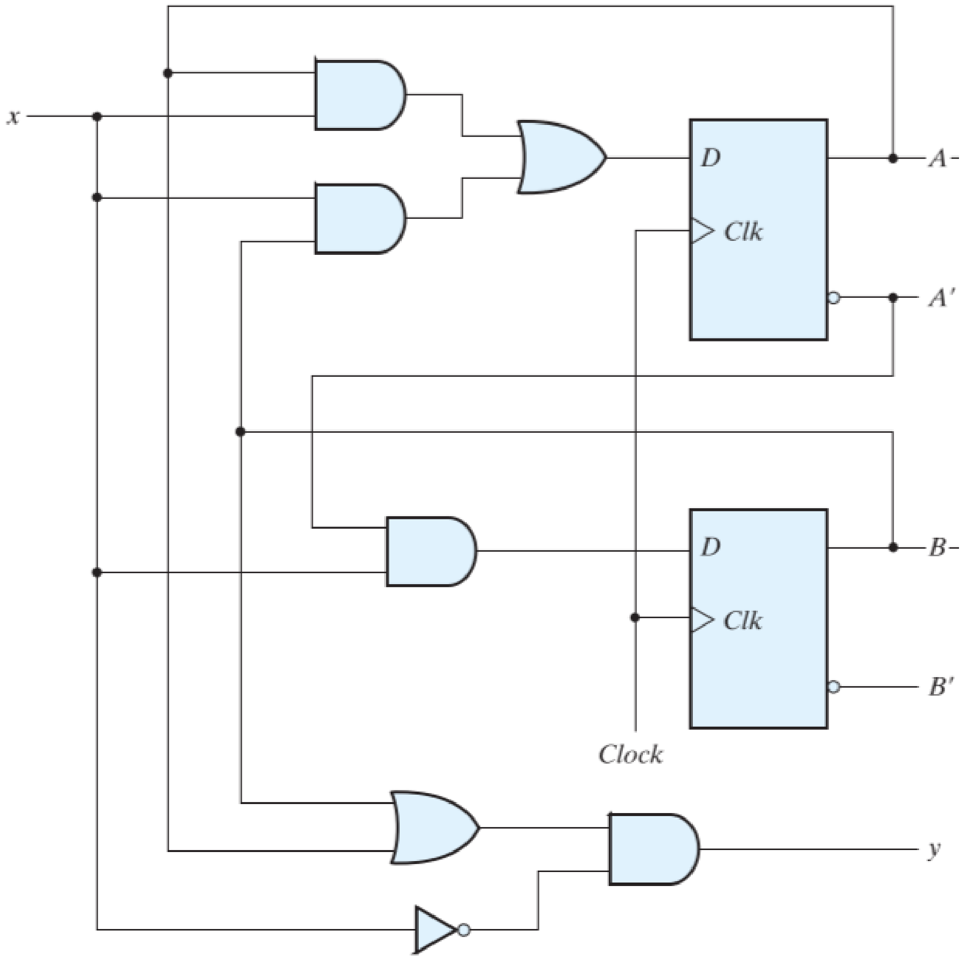
Q(t)	Q(t + 1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

May 26

Analysis of synchronous sequential circuits

The idea is to find what the circuit does (in the form of say a state diagram) given the circuit diagram

Eg.

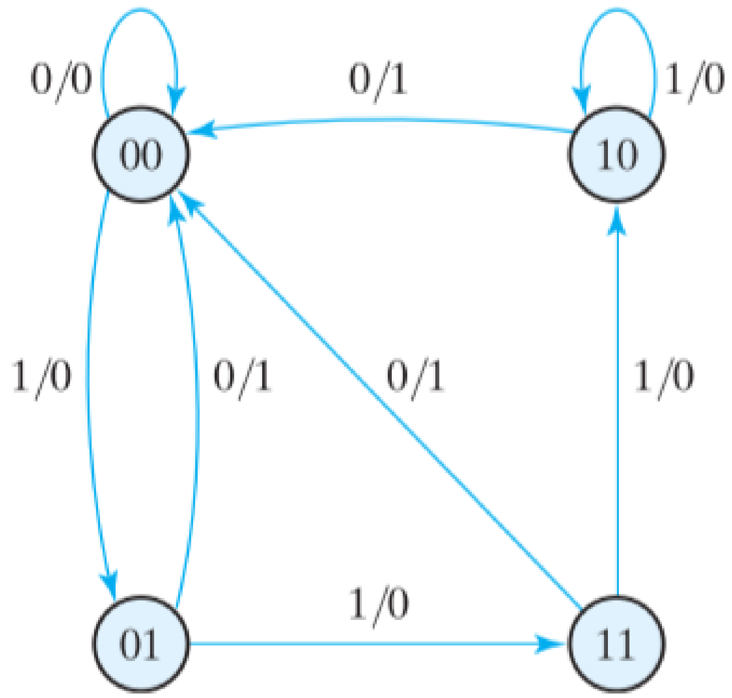


$A_{t+1} = A_t x_t + B_t x_t$
$B_{t+1} = A'_t x_t$
$y = (A_t + B_t)x'_t$

State Table:

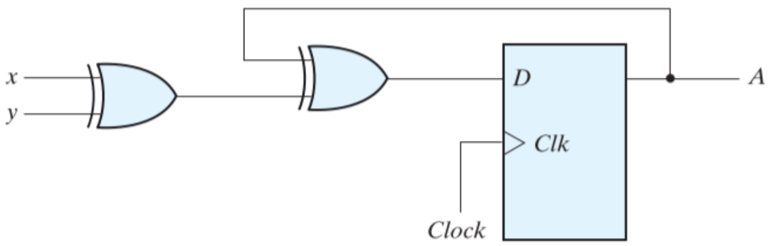
Present State		Input	Next State		Output
A	***B***	***x***	***A***	***B***	***y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

State Diagram



This circuit detects zeroes after a series of ones

Eg.

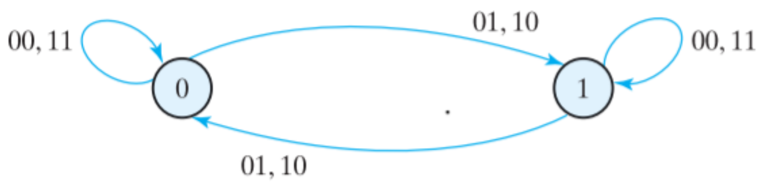


$A_{t+1} = A \oplus x \oplus y$

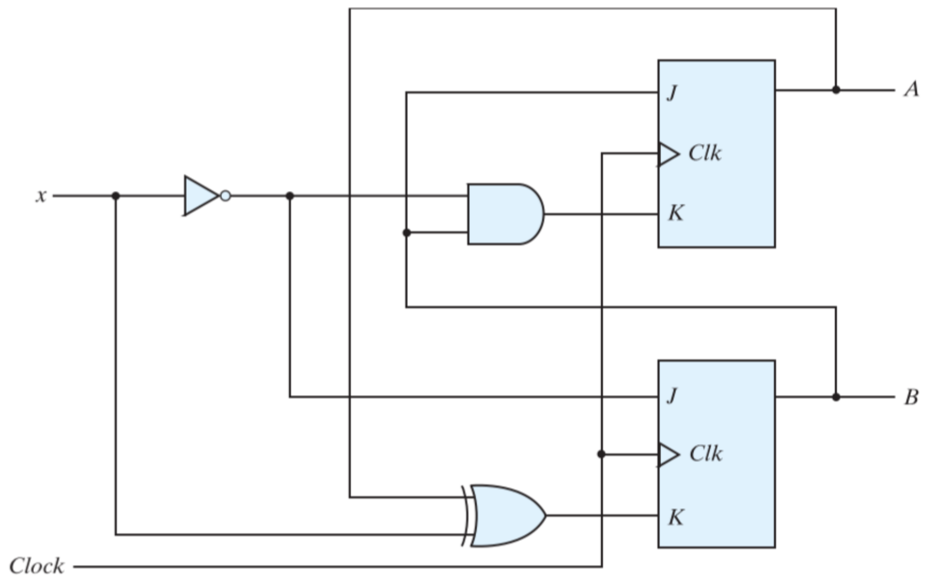
State Table

A	x	y	A_{t+1}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

State Diagram



Eg.

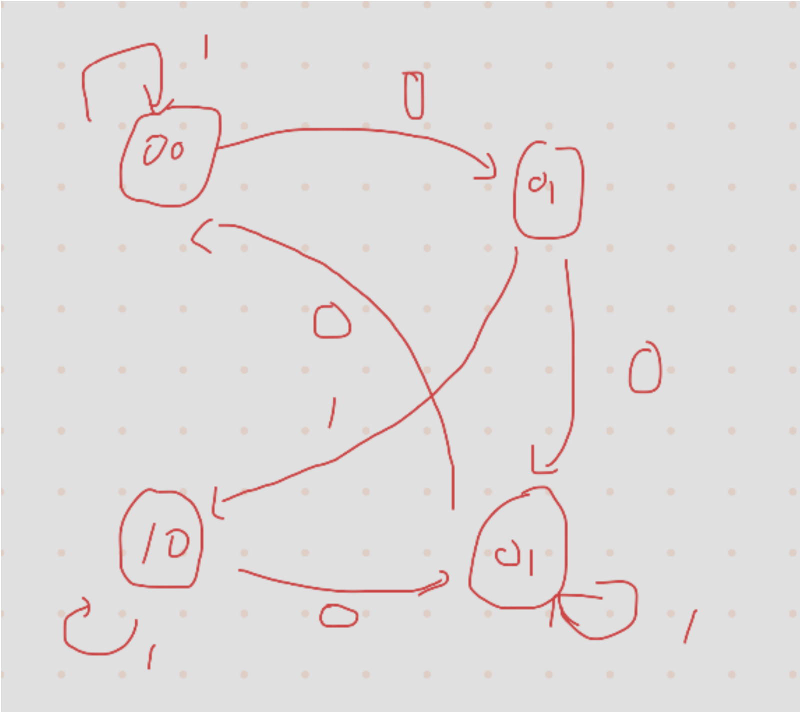


$J_A = B$
$K_A = Bx'$
$J_B = x'$
$K_B = A \oplus x$

State Table

A_t	B_t	**x**	A_{t+1}	B_{t+1}	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

State Diagram

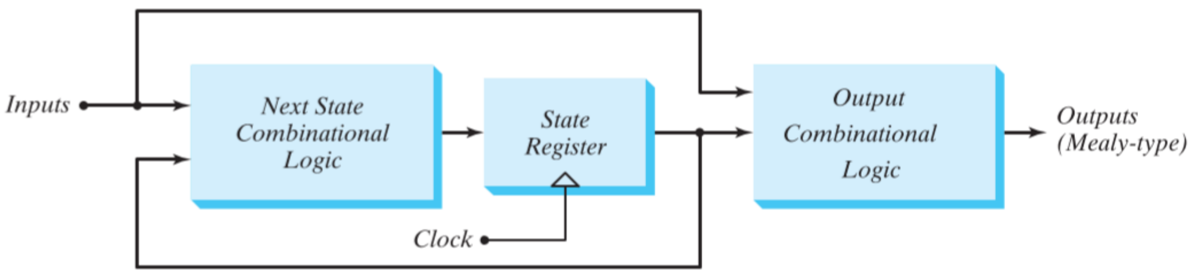


Finite State Machines:

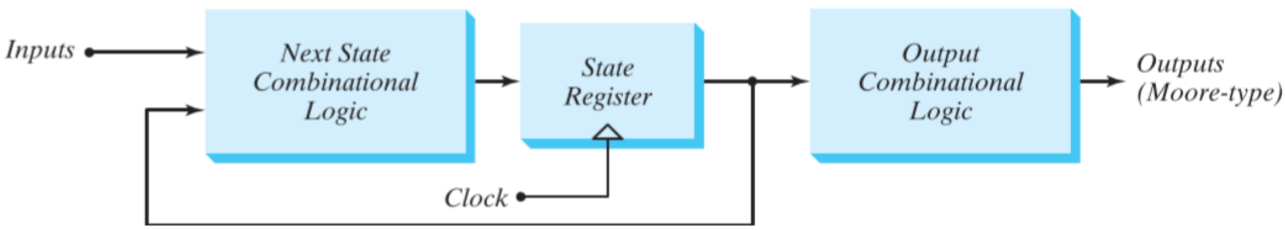
Mealy Machine

- Output depends directly on input

Mealy Machine



Moore Machine:

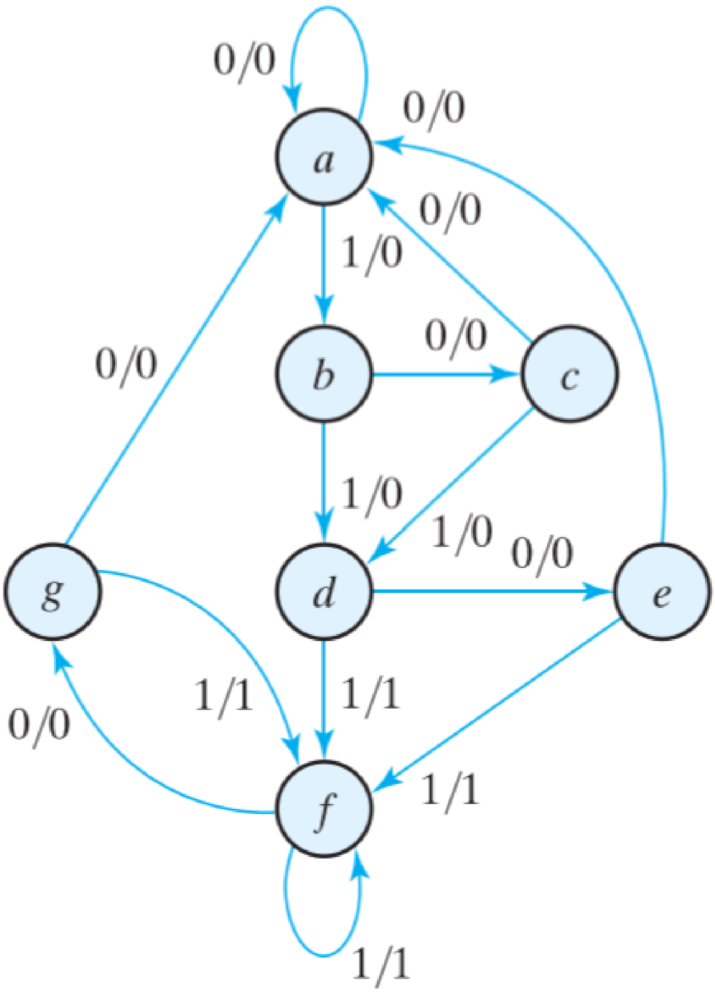


- Output doesn't depend directly on input

May 27

State Reduction

Eg.



Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

There are some redundant terms here. For example, e and g do the same thing, so we can get rid of it. If we set e = g, d and f also turn out to be the same.

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

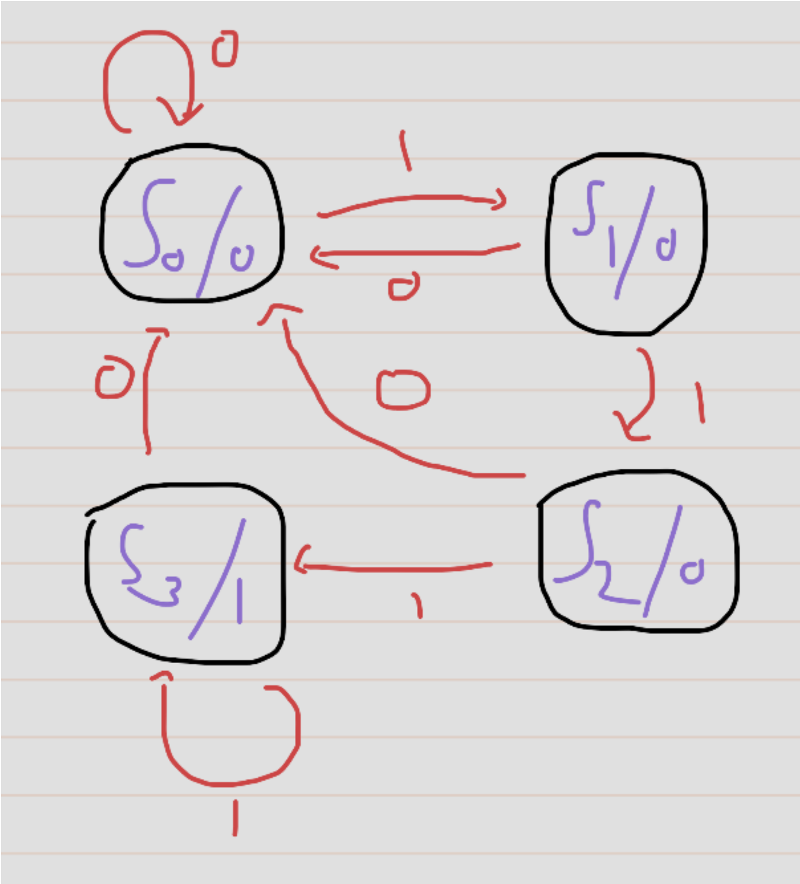
Design Procedure

- Derive state table
- Reduce number of states
- Assign unique binary value for each state
- Obtain binary coded state table
- Choose type of flip flops to be used
- Derived simplified flip-flop input equation
- Draw logic circuit

Two bit counter

4 states are required

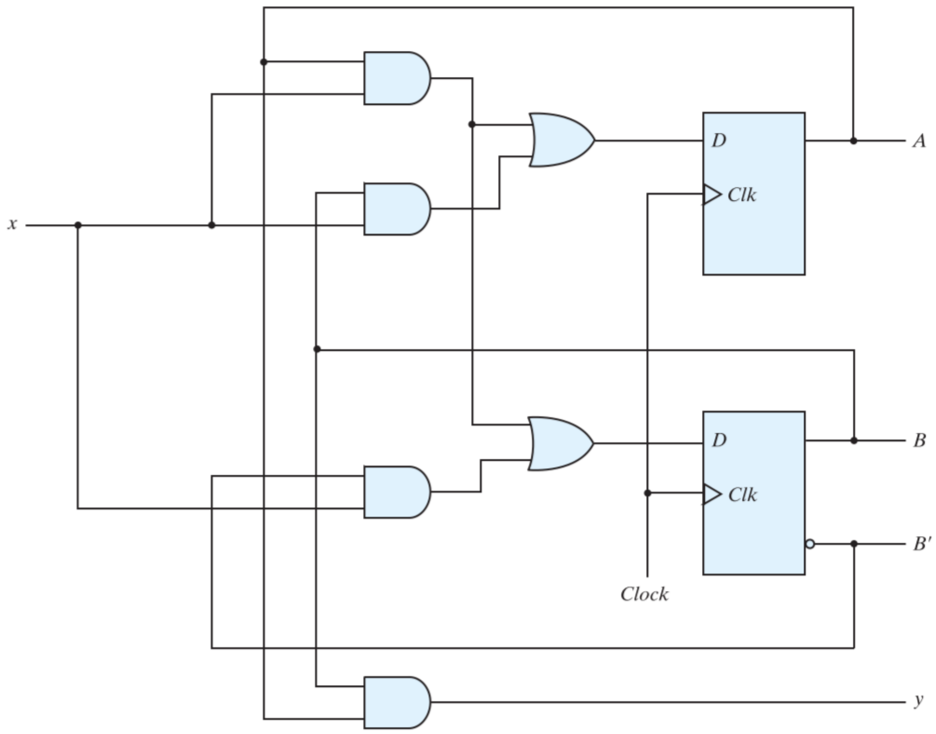
- 2 bits/flip-flops
- 1 input



A_t	B_t	**x**	A_{t+1}	B_{t+1}	**y**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

$A_{t+1} = x(A_t + B_t)$
$B_{t+1} = x(A_t + B_t')$
$y = AB$

Synthesis using D Flip-flop



Synthesis using JK Flip-flop

$$A_{t+1} = x(A_t + B_t)$$
$$B_{t+1} = x(A_t + B_t')$$
$$y = AB$$

State Table

Table 5.13
State Table and JK Flip-Flop Inputs

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

May 30

3-bit Counter

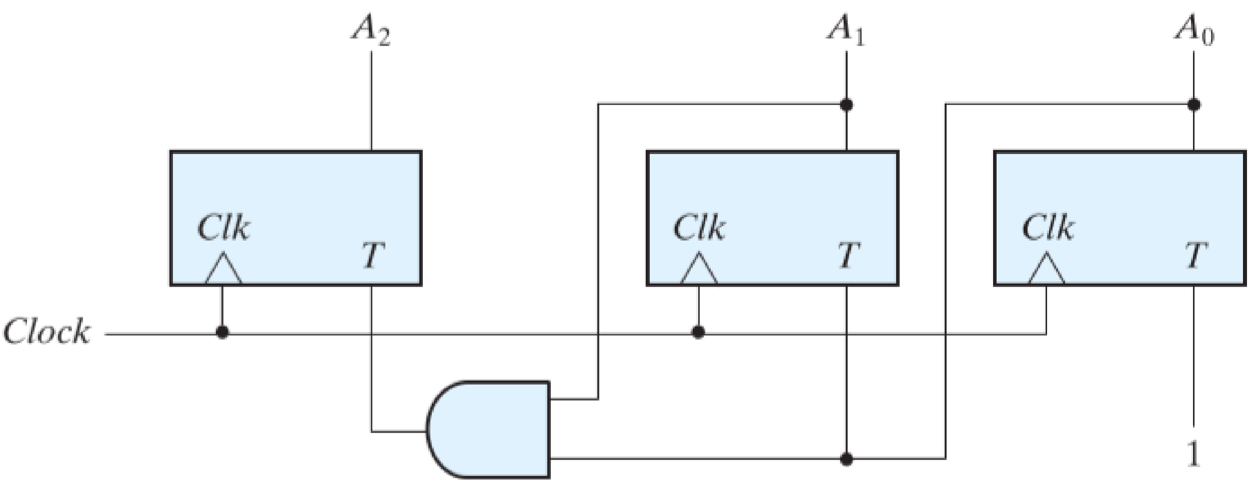
Synthesis using T flip-flop

Excitation Table

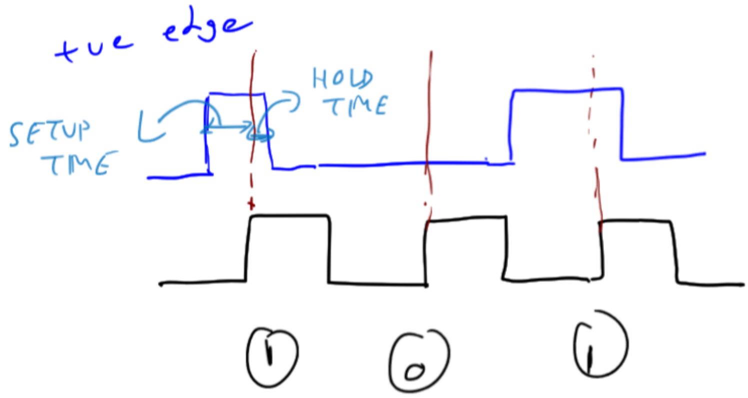
State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

$T_{A_0} = 1$
$T_{A_1} = A_0$
$T_{A_2} = A_1 A_0$



Timing Circuits

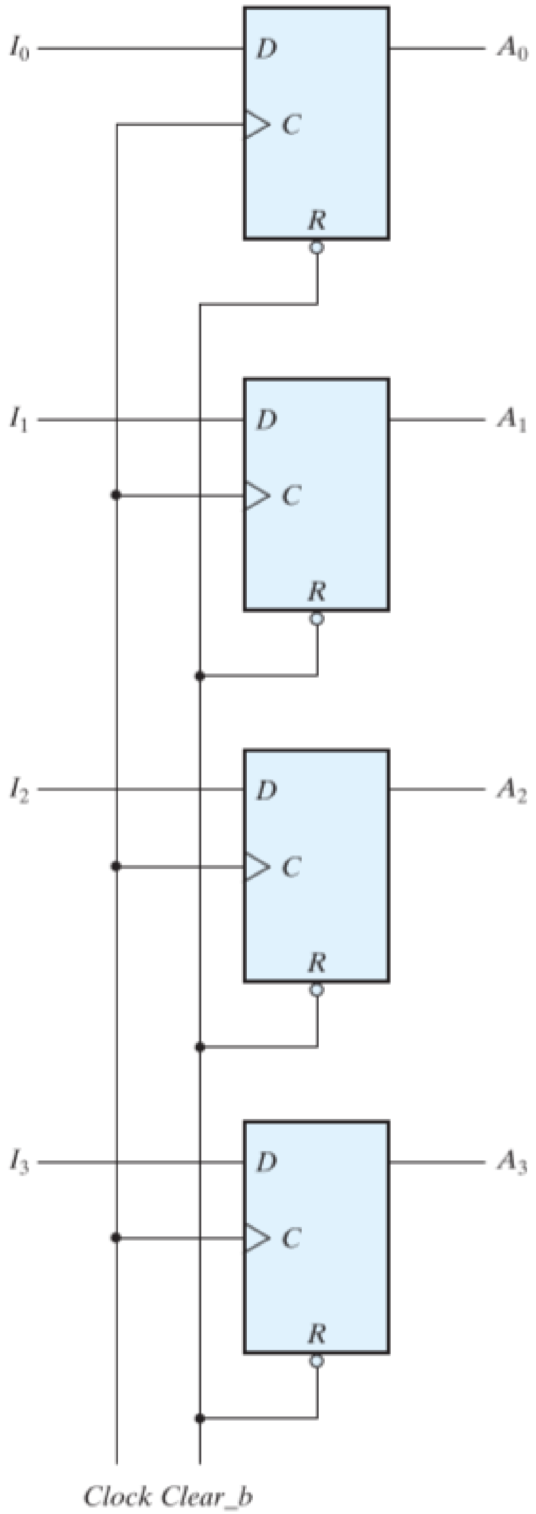


Setup time is defined as the minimum amount of time before the clock edge, during which the data must be held stable i.e. the input should not change within the setup time duration so as to ensure proper data to be latched.
Hold time is defined as the minimum amount of time after the clock edge, during which the data must be held stable i.e. the input should not change within the Hold time duration so as to ensure proper data to be latched.

Registers and Counters

Registers are a group of flip-flops and gates that store data

Asynchronous register using D flip-flops



- I0, I1, I2, I3 are the inputs stored
- Clear_b
  - is 1 by default
  - is 0, all flip-flops are set to 0

× **Problems:**

The output changes during clock-edge if inputs are changed. There are two ways to "store" the information:

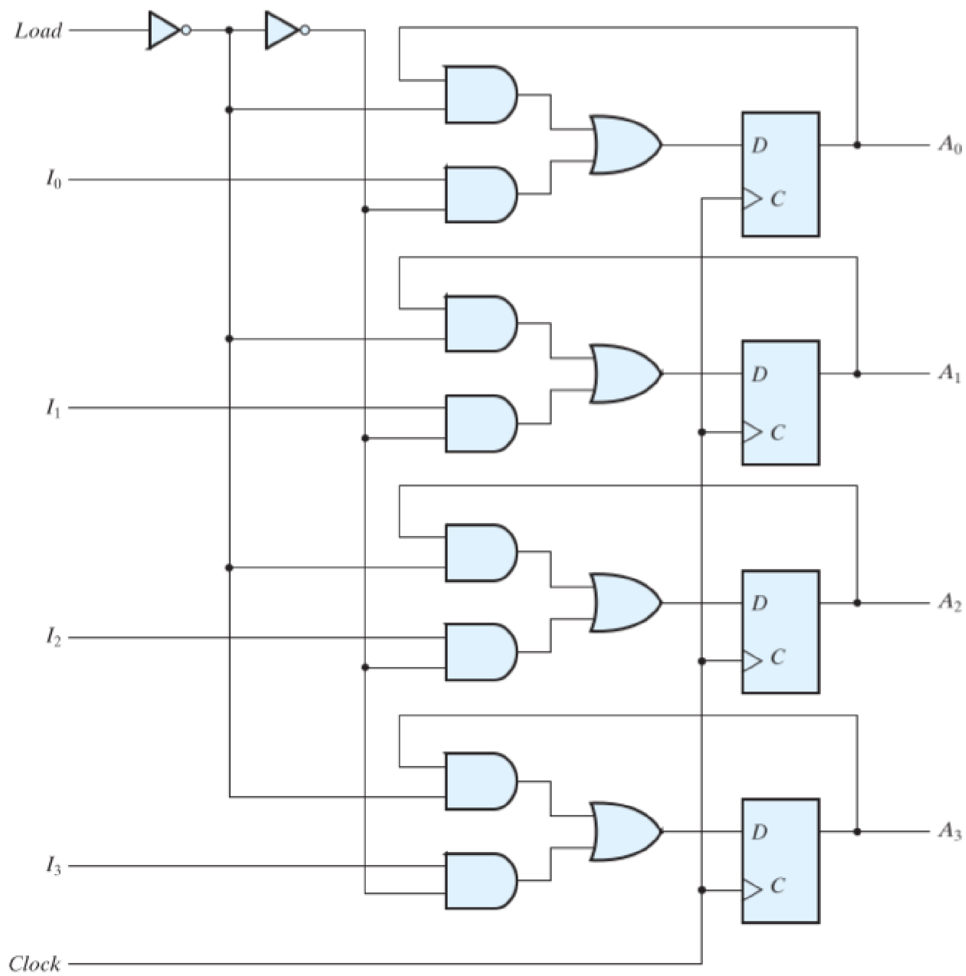
- Clock should be held in pause
- Input should be held constant

Messing around with either of them is not a good idea!

Synchronous register

4-bit parallel load registers

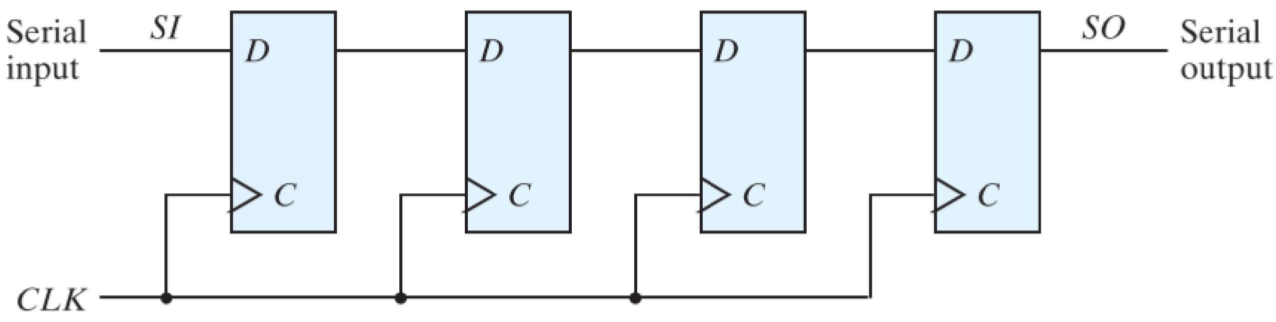
Holds the output in a given state until you are ready to change the state ("loading")



- When load is 1, Input data is transferred
- When load is 0, existing data is stored undisturbed

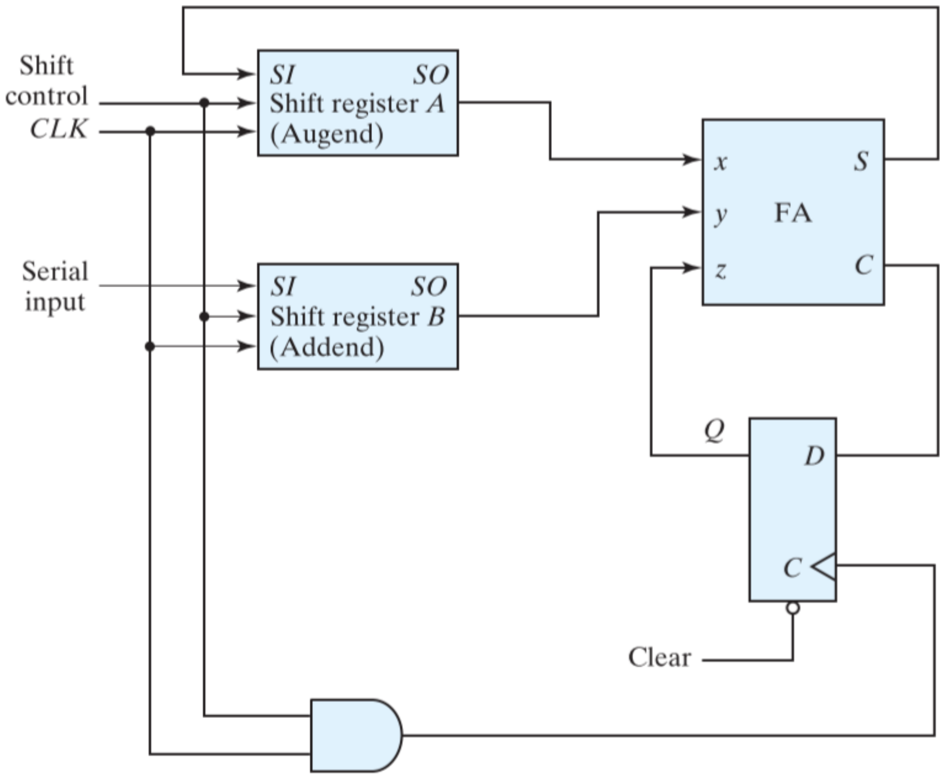
Shift Register

Shifts data to neighbouring flip-flops during each clock cycle



Serial Adder

4-bit serial adder by loading through shift resistor and storing sum in another shift register



Present State Q_t	Inputs : x	y	Next State Q_{t+1} = C	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

$J_Q = xy$
$K_Q = x'y'$
S = $x \oplus y \oplus Q_t$

June 2

Bidirectional Universal Shift register

- Parallel/serial in, parallel/serial out
- Shift from left to right and right to left

S_0	S_1	**Operation**	
0	0	No change	$Q_{t+1} = Q_t$
0	1	Shift right	$A_3 \to A_2 \to A_1 \to A_0$
1	0	Shift left	$A_0 \to A_1 \to A_2 \to A_3$
1	1	Parallel load	Parallel input

- Clear is used to asynchronously reset the D flip-flops

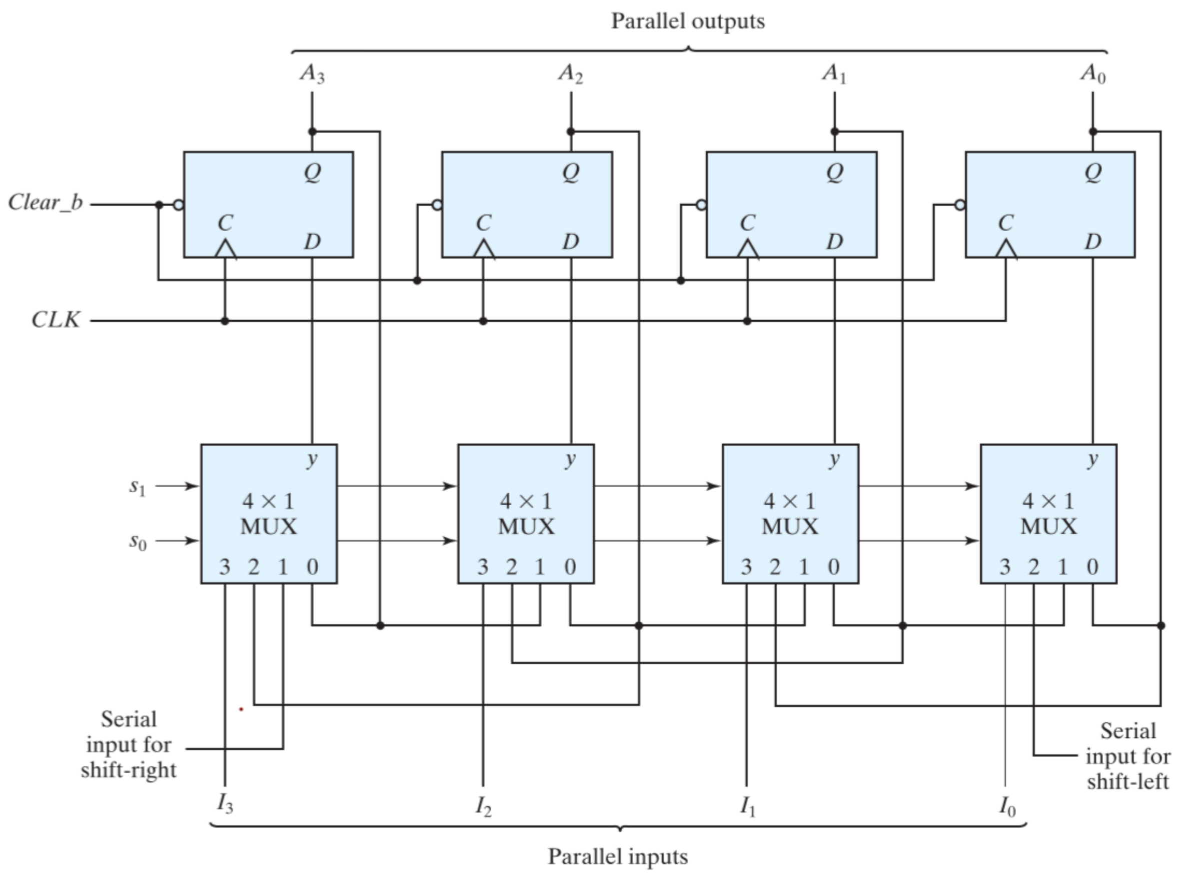
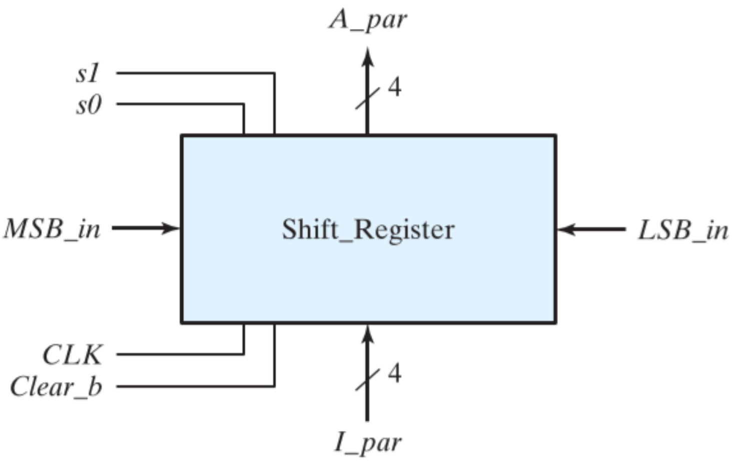


Diagram:



June 3

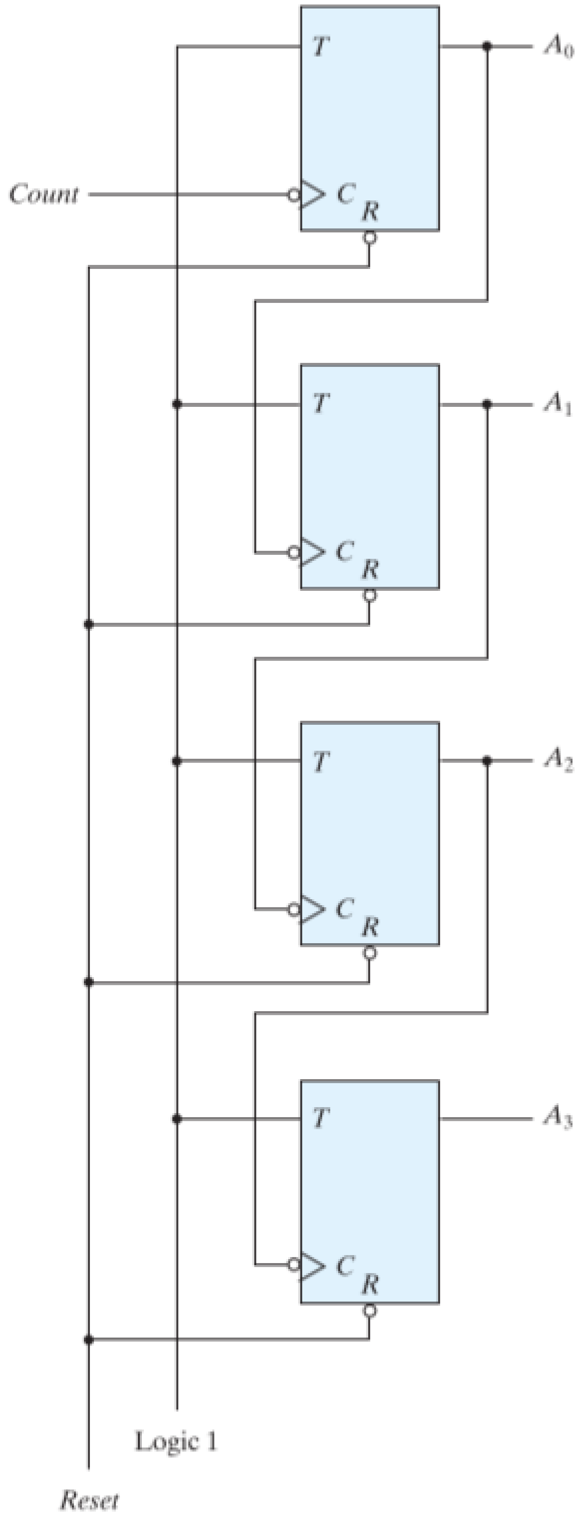
Counters

Subset of register, in which it goes through pre-defined sequence of binary states

Asynchronous

Ripple counter

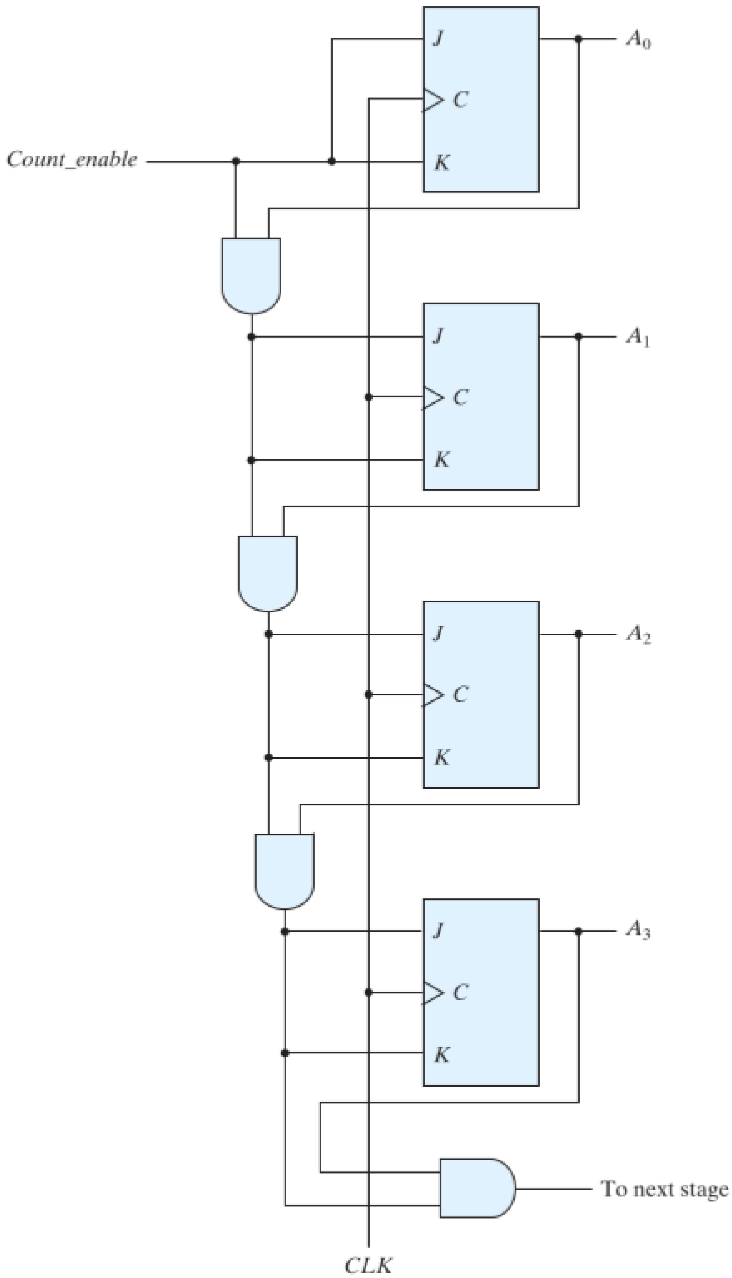
- flip-flop transition triggers the next flip-flop (it's not synchronized by a common clock)



Synchronous counter

- Flip flops triggered by common clock

Up-Down counter
Use T flip-flop instead of JK flip-flop



BCD Counter using T flip-flop

A3	A2	A1	A0	A3	A2	A1	A0	y	T_{A3}	T_{A2}	T_{A1}	T_{A0}
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

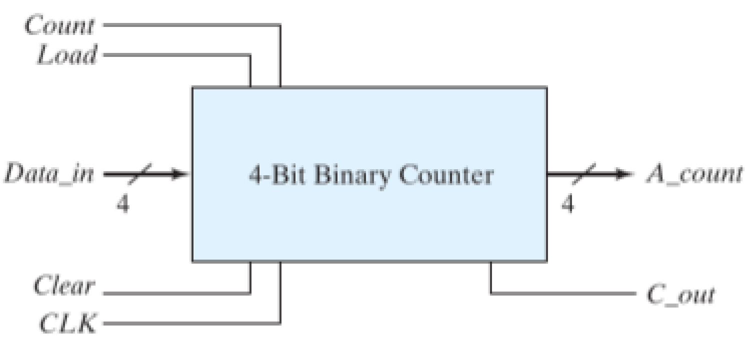
A3	A2	A1	A0	A3	A2	A1	A0	y	T_{A3}	T_{A2}	T_{A1}	T_{A0}
1	0	0	1	0	0	0	0	1	1	0	0	1

T_{A0} = 1
T_{A1} = A3' A0
T_{A2} = A3' A1 A0 (can be simplified further)
T_{A3} = A3' A2 A1 A0 + A3 A2' A1' A0 (can be simplified further)
y = A3 A2' A1' A0

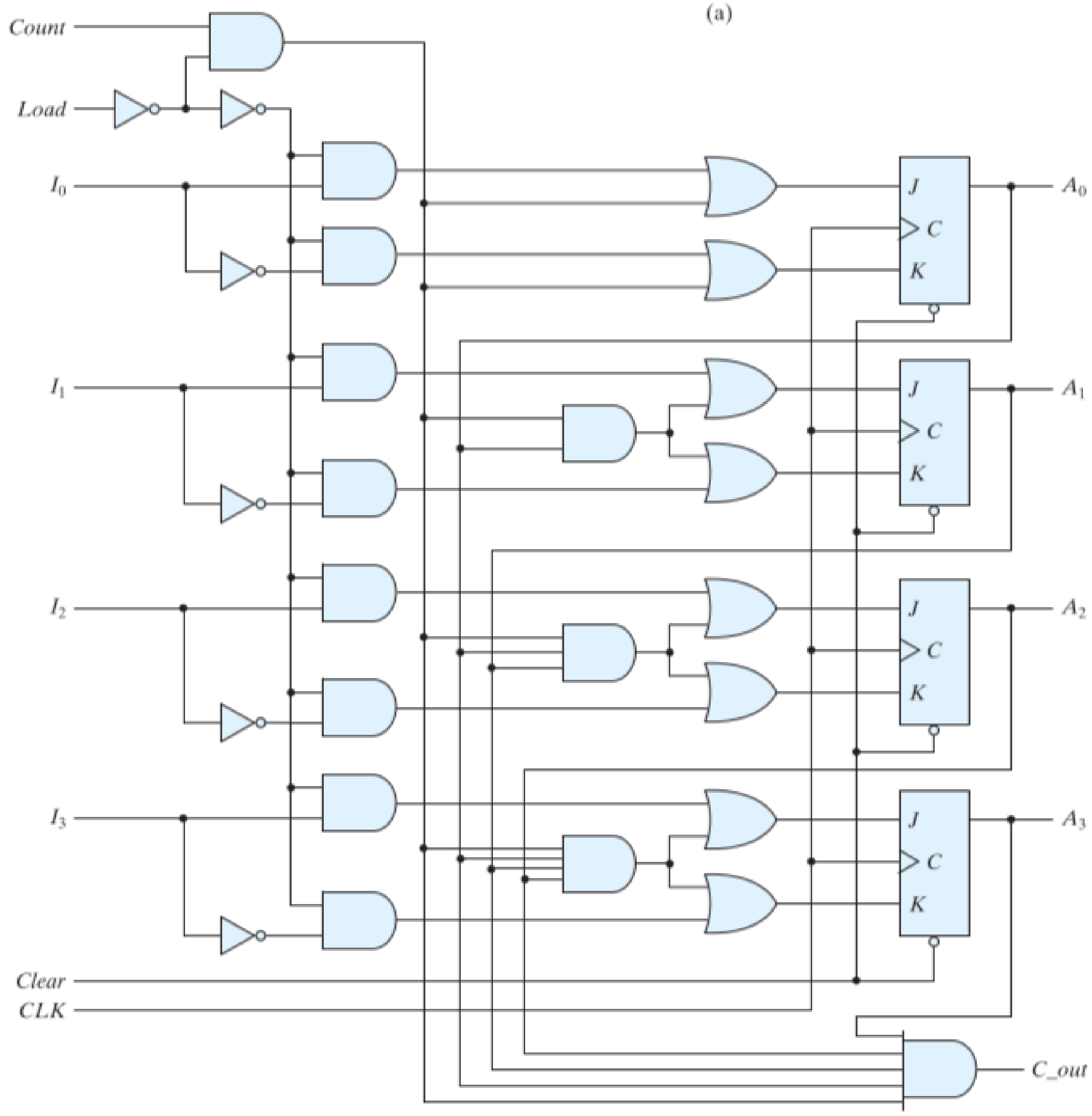
Binary counter with parallel load

- We want parallel loading capacity, as we often want to start the counter at a particular state instead of some arbitrary state

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

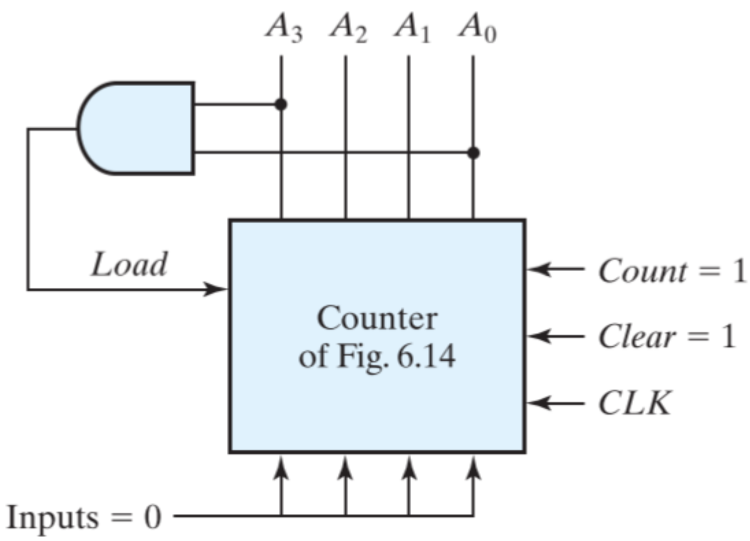


(a)

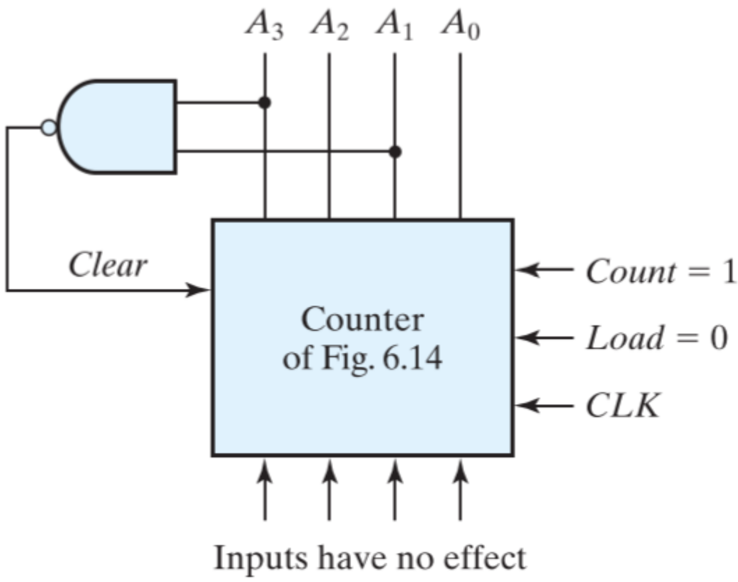


(b)

Eg. BCD counter with parallel load

- 

  All inputs are 0, when A3 A0 are like 11, we reach 1001 we load the counter with 0000

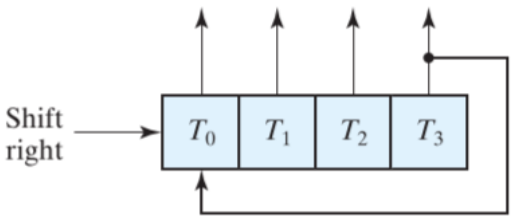
- 

  When A3 A1 are like 11, we've hit 1001 and we clear the counter

Ring counter
Inputs going like 1000, 0100, 0010, 0001, 1000, etc.

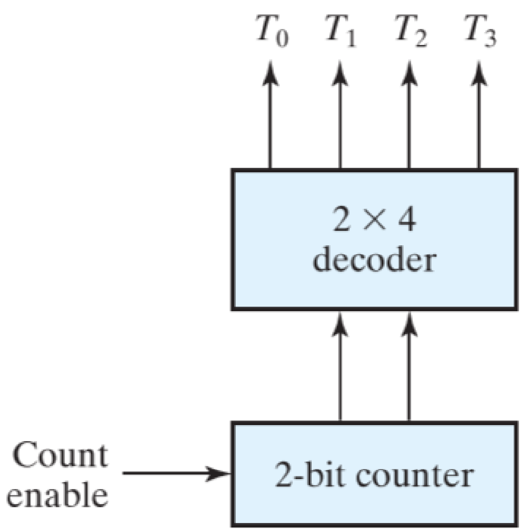
For 2^n timing circuit

Shift register: Shift input is T3



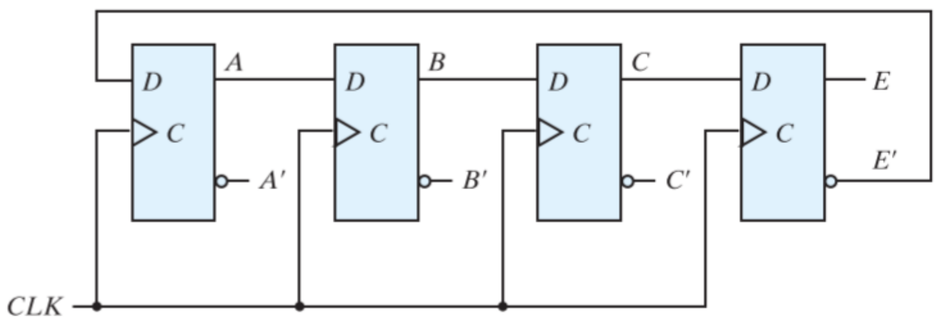
- 2^n flip-flops

2-bit counter and decoder



n flip-flops and 2^n four input and gates

Johnson Counter/Twisted ring counter/Switch tail counter



Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

For n timing sequences, there are n/2 flip-flops and n gates required, and all are two input gates

June 6

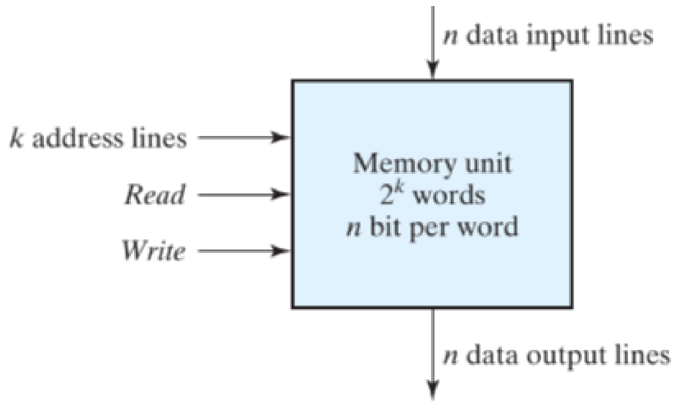
Memory Units

Word: Groups of bits stored by memory units
Modern computers are 64-bit computers
Address: Integer from $0 \to 2^k - 1$, where k is the number of address lines

Read-Only Memory	Random Access Memory
Non-volatile, i.e. doesn't lose information when powered down	Volatile
Hard-wired (look-up table)	Faster than a ROM
"Read" only	Both read and write

Random access memory

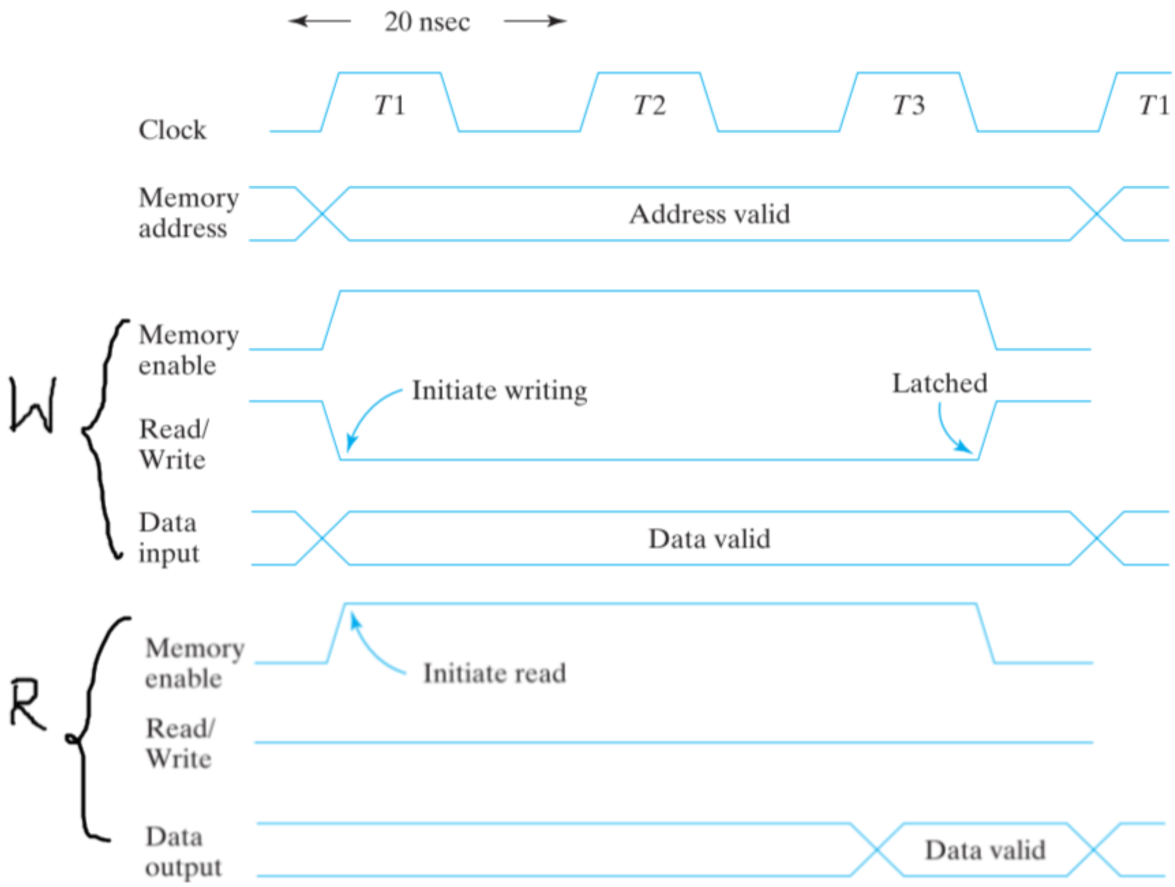
Diagram of a memory unit and its functionality:



Memory Enable	Read/Write	Operation
0	X	None
1	0	Write
1	1	Read

→ Access time: Time required for read operation
→ Cycle time: Time required for write operation
Access time and cycle time < Clock time of CPU

Clocked at 50 MHz, i.e. Time period of one pulse is 20 nsec

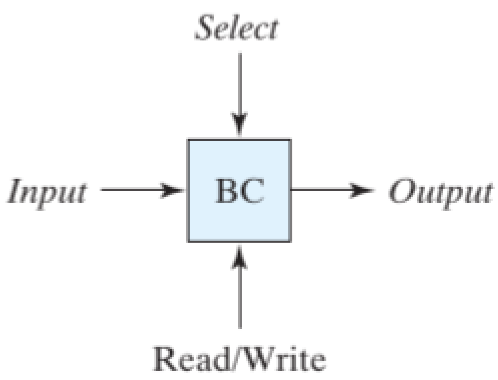


SRAM	DRAM

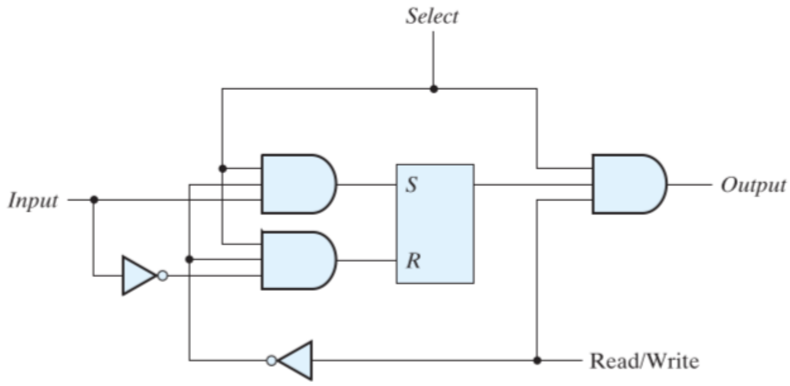
SRAM	DRAM
Static RAM	Dynamic RAM
Latches and flip-flops	Capacitor, needs refreshing
Retains data for a long timescale	Shorter timescale
Faster	Slower
Expensive	Less expensive
More power	Less power
Address multiplexing is more complicated, as it uses 4 transistors	Address multiplexing is easier, as it uses transistor + capacitor

Memory Cell

- Stores one bit of information
- For a RAM with m words and n bit word length, there are m x n memory cells

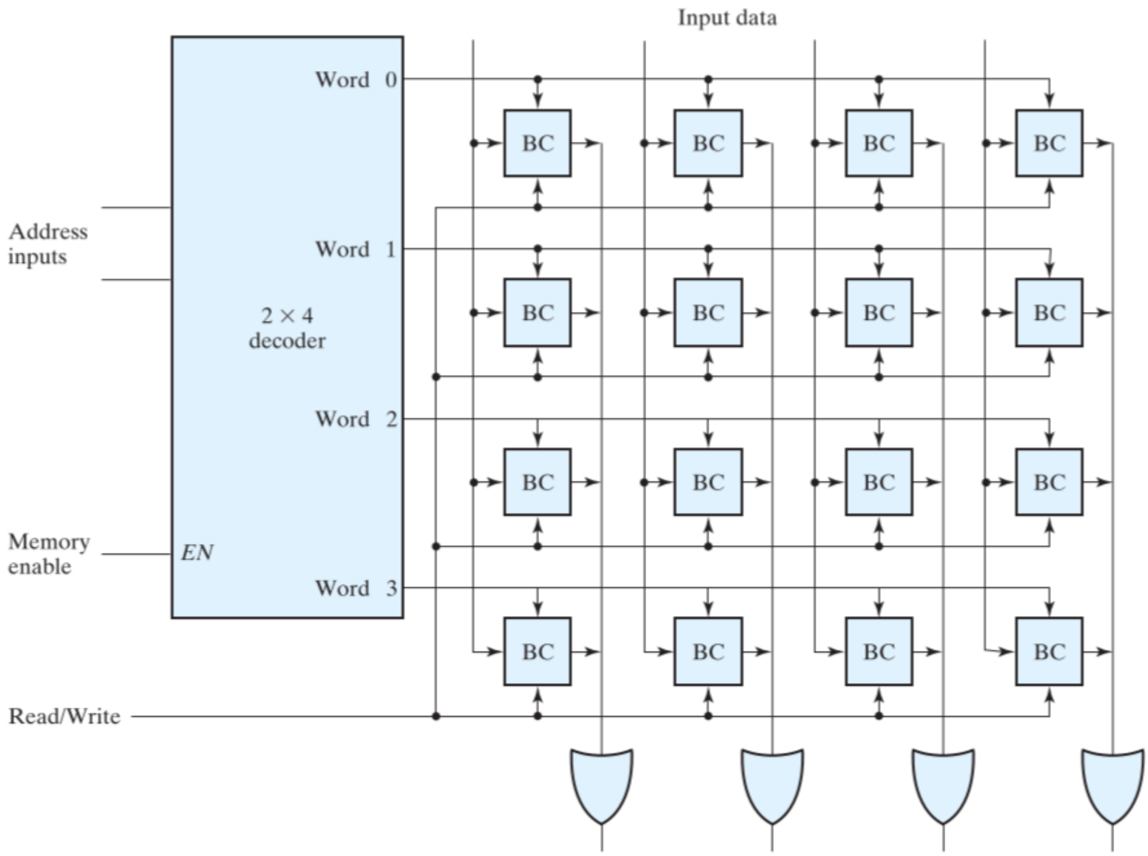


Select	Read/Write	Input	Output	Q_+
0	X	X	0	Q
1	0	I	0	I
1	1	X	Q	Q



(a) Logic diagram

Construction of 4 x 4 RAM

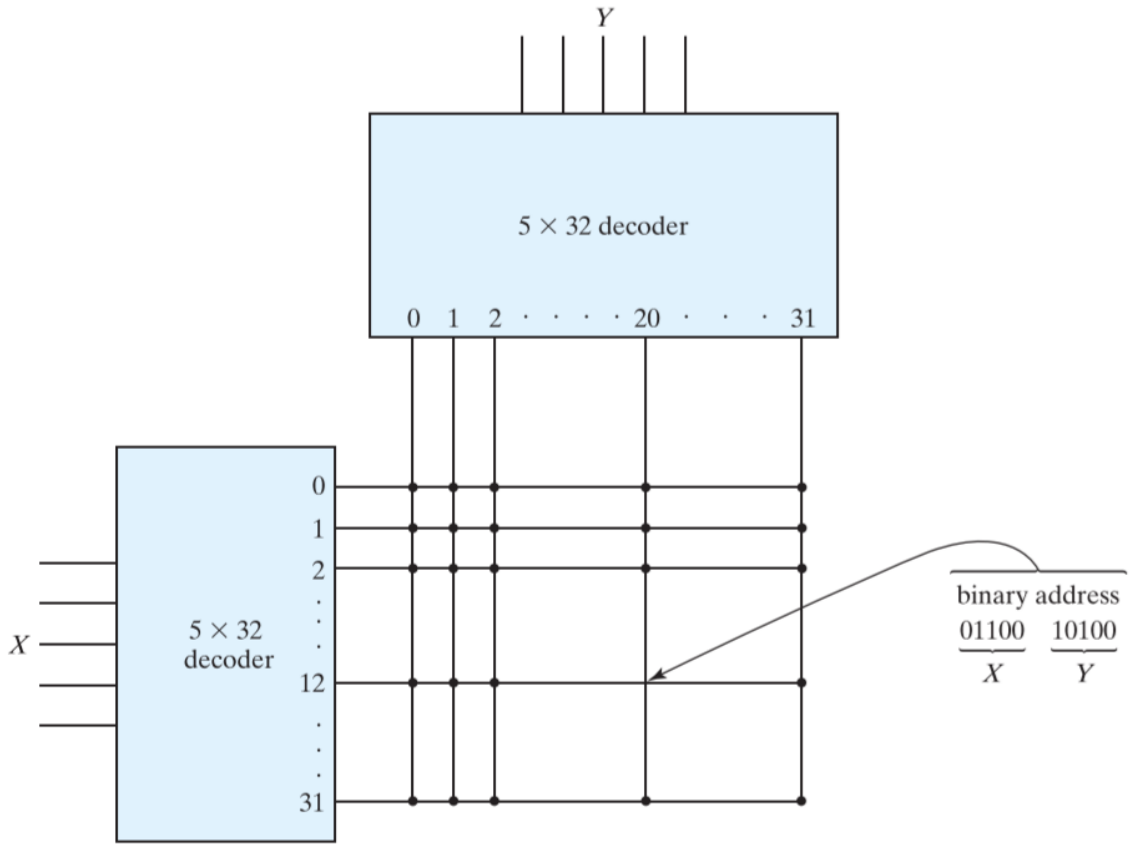


> **Issues**
>
> k inputs and 2^k words
> The decoder requires 2^k AND gates with k inputs (k + 1, if we have the enable pin) each

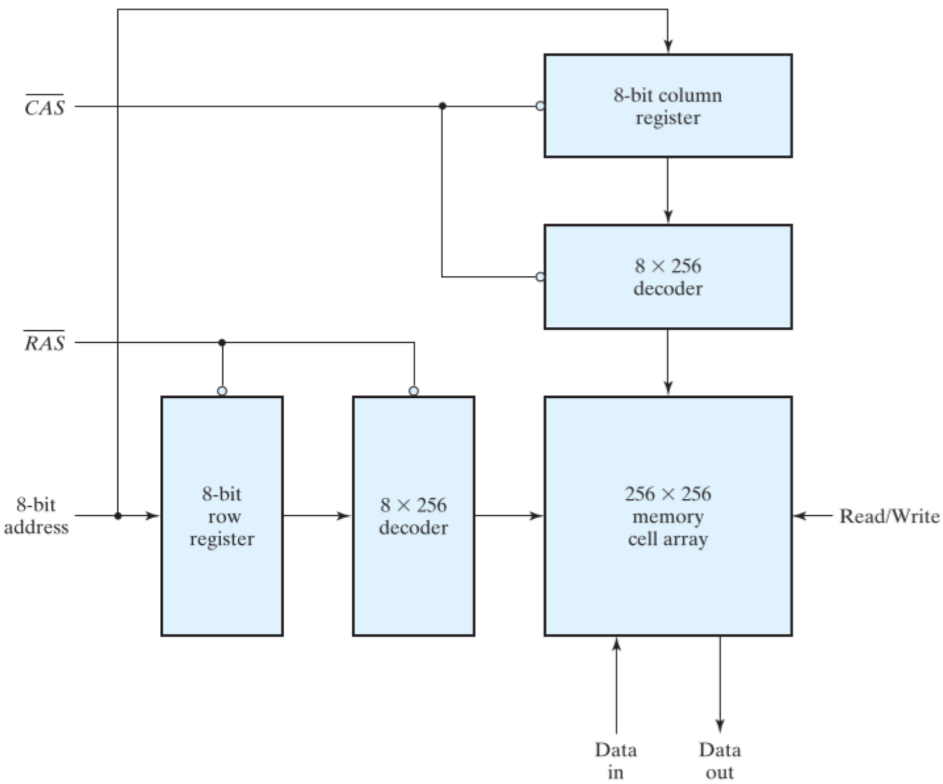
Coincident decoding
Keep track of x and y coordinates, using $k/2$ bits for x coordinates and $k/2$ bits for y coordinate.

This uses 2 * (32 5-input AND gates) and the memory cell is at the intersection of these two rows and columns.

1024 words memory, using coincident decoding



Address Multiplexing in DRAM



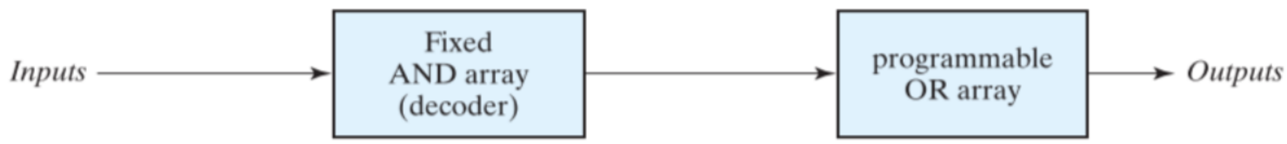
Strobes: Essentially enable pins, that enable the data in register to the decoder when 0

- RAS: Row address strobe
- CAS: Column address strobe
  First, RAS is made 0, a particular row is chosen, then CAS is made 0, the respective column is chosen, then the data is read/written. After that, both strobes are reset to 1

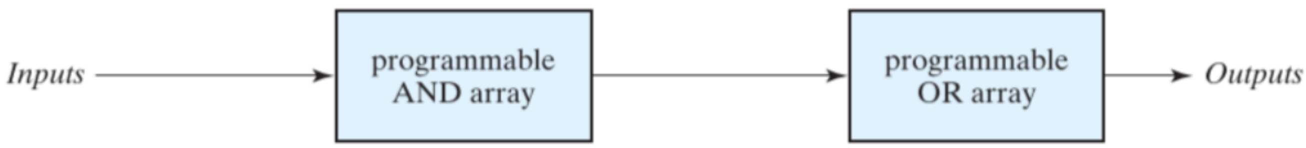
Programmable Logic Devices

- I/O Block → Programmable switch matrix → I/O Block
- Implemented using AND-OR gates

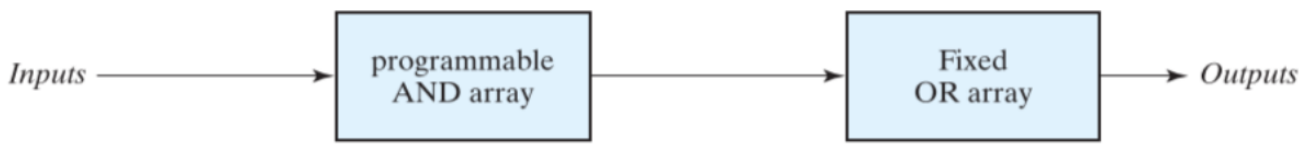
Programmable Read Only Memory:



Programmable Logic Array:

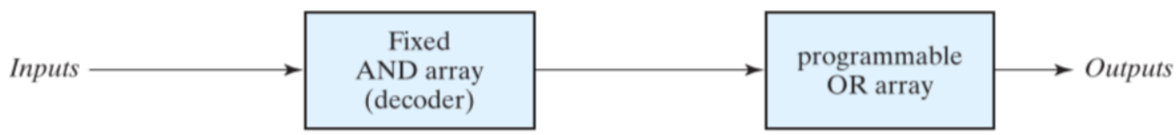


Programmable Array Logic:



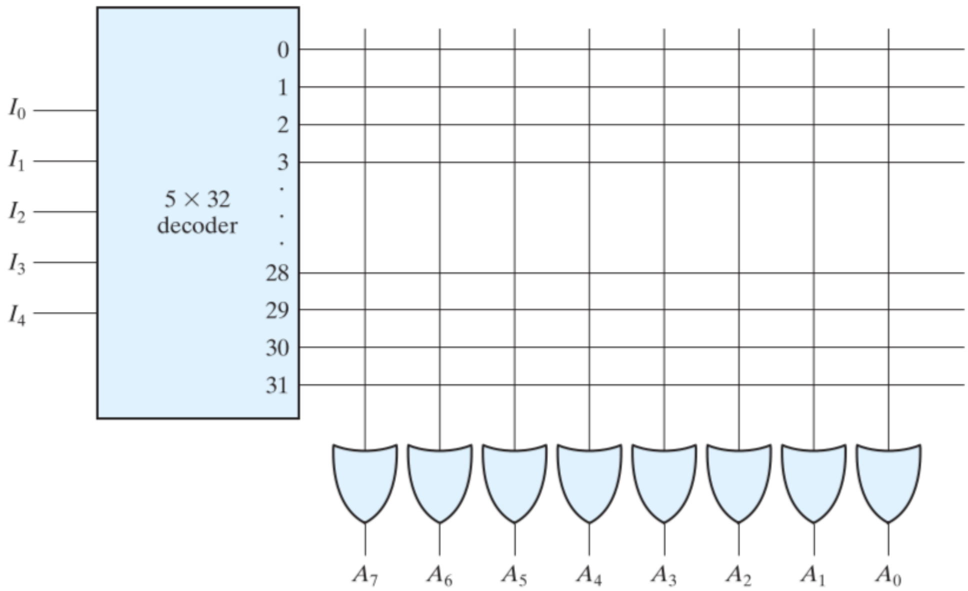
Programmable Read only memory (PROM)

- Memory elements are interconnections patterns along with decoder/OR gates
- 2^k words, each word is n bits long



- Programmable OR array is built with fuses intact, fuses broken while programming the device

Eg. 32 × 8 PROM



Note: Each OR gate is a 32 input OR gate

Sample truth table

Inputs					Outputs							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮							⋮			
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

Using PROMs to design functions

- Each of $A_0 \ldots A_7$ can be used as a function

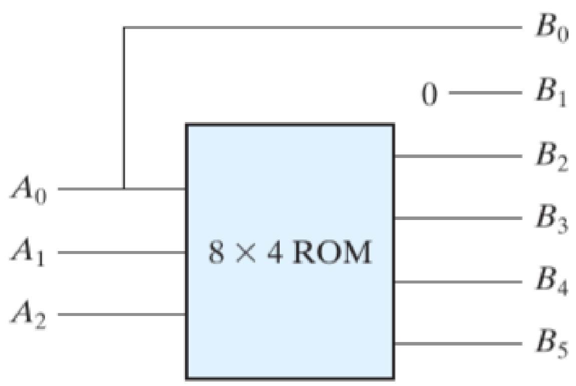
Eg. Design a combinational circuit using a programmable read only memory to accept a 3-bit number and outputs a binary number equal to square of the input number

A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0

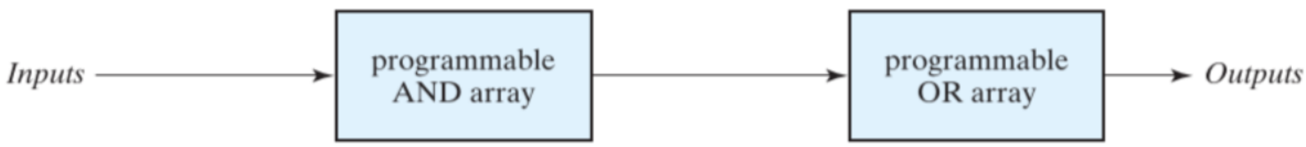
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	1

- $B_0 = A_0$
- $B_1 = 0$
- $B_2 \ldots B_5$ all depend on the input, so they the word length has to be 4

ROM would be an 8×4 ROM, where word length is 4 and we have to address 8 locations. We can burn appropriate connections corresponding to $B_5 \ldots B_2$ in the above table



Programmable Logic Array (PLA)



Both AND gates and OR gates are programmable

Three parts:

- AND gate inputs are programmable
- OR gate inputs are also programmable
- Complemented outputs can be XOR-ed to get proper output
- Using common outputs is better

PLA size:

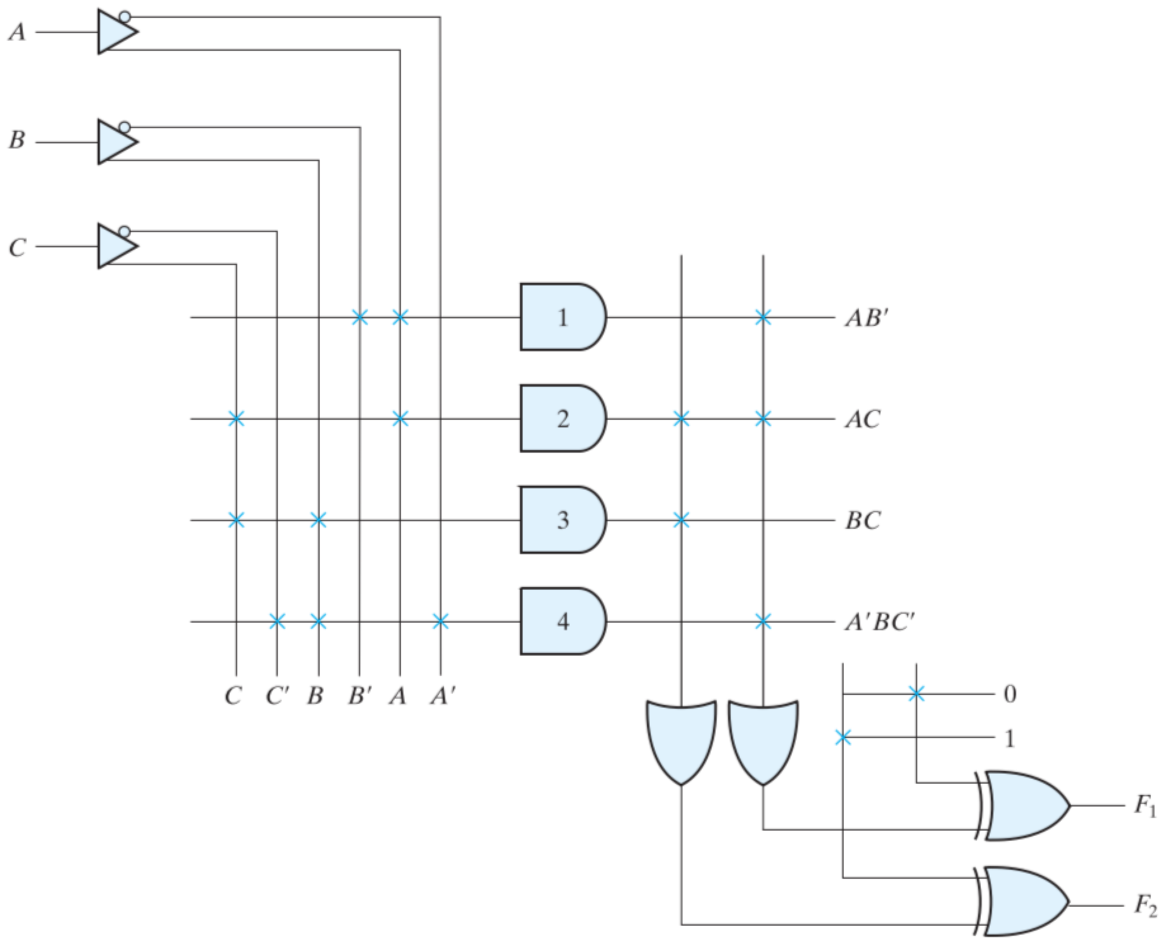
- n inputs
- k 2n-input AND gates
- m Outputs
- $2n \times k$ Number of connections between input and AND
- $m \times k$ connections between AND and OR

Typical PLA size would be $16/48/8 \rightarrow n/k/m$

Eg.
$F_1 = AB' + AC + A'BC'$
$F_2 = (AC + BC)'$

Programming Table

A	B	C		$F_1(T)$	$F_2(C)$
1	0	-		1	_
1	_	1		1	1
_	1	1		_	1
0	1	0		1	_



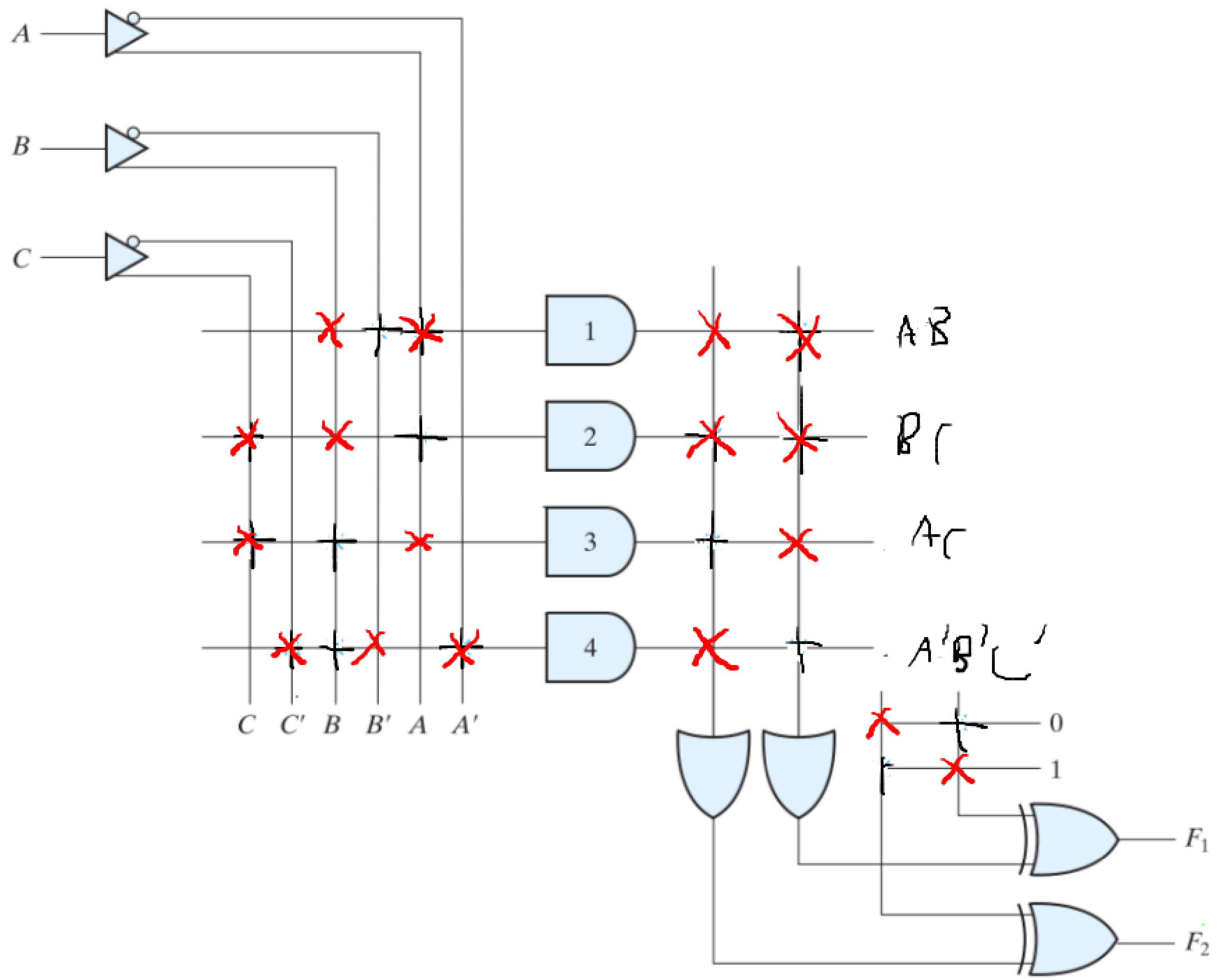
Eg.
F_1 = $\sum$ (0, 1, 2, 4)

$F_2 = \sum (0, 5, 6, 7)$

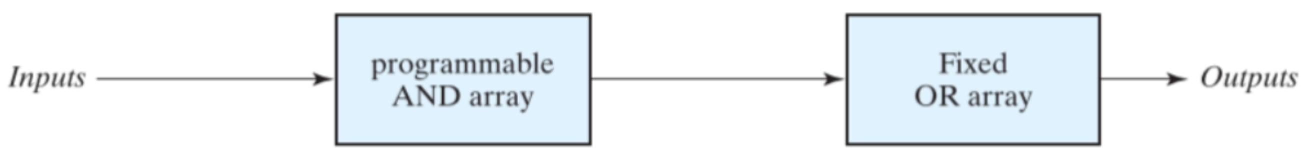
$$F_1 = A'B' + B'C' + A'C'$$
$$F_1' = AB + BC + AC$$
$$F_2 = A'B'C' + AC + AB$$

Programming Table

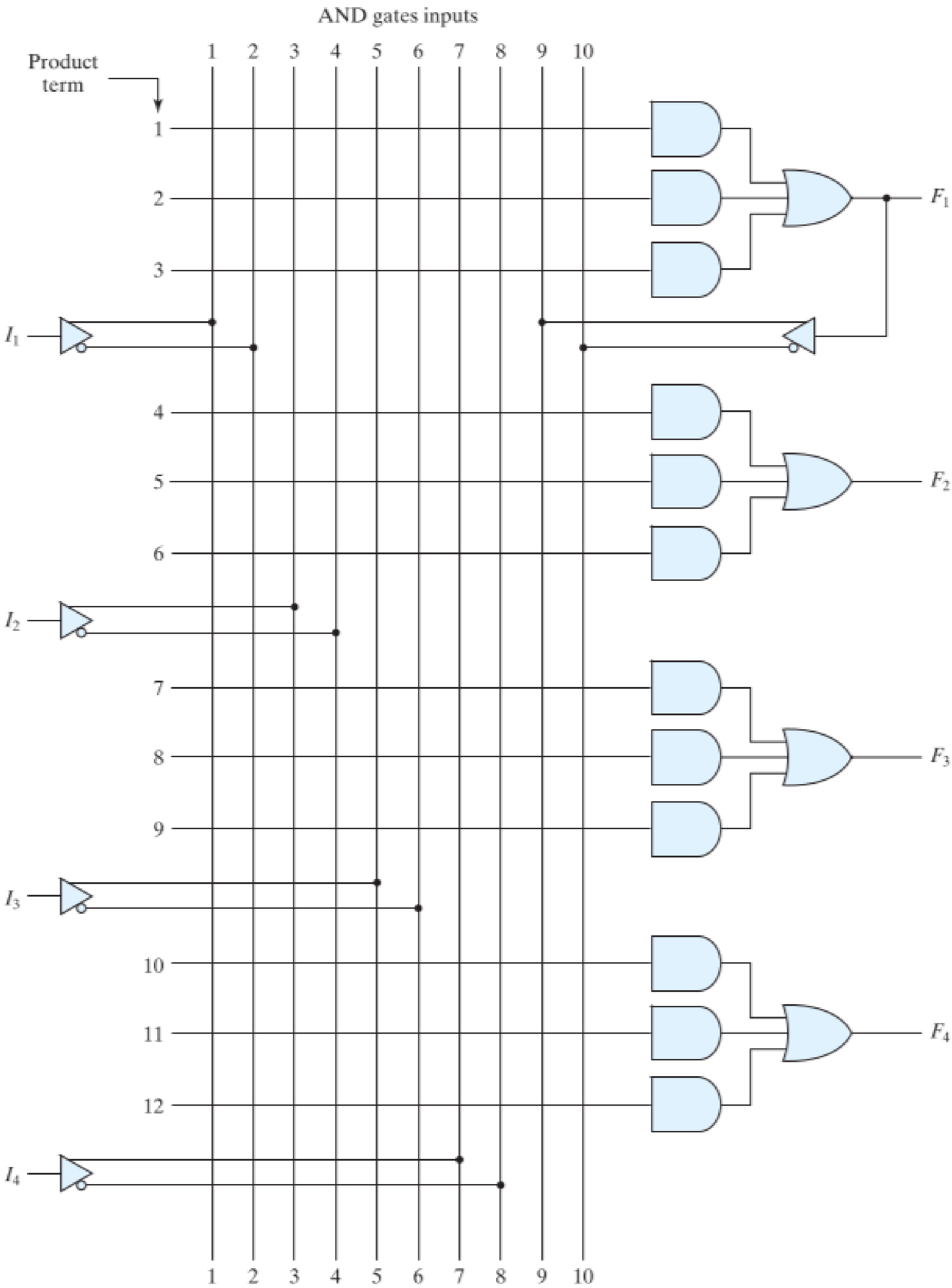
A	B	C		$F_1(C)$	$F_2(T)$
1	1	-		1	1
1	_	1		1	1
_	1	1		1	_
0	0	0		_	1



Programmable Array Logic



- Fixed OR array (not as flexible as PLA)
- Typical 4 inputs and 4 outputs with 3 AND + OR gate at each output



- Lines $1 \dots 8$
- Lines 9 and 10 can be used to feed in the first input

Eg.
W = $\sum$ (2, 12, 13)
X = $\sum$ (7, 8, 9, 10, 11, 12, 13, 14, 15)
Y = $\sum$ (0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)
Z = $\sum$ (1, 2, 8, 12, 13)

$W = A'B'CD' + A B C'$

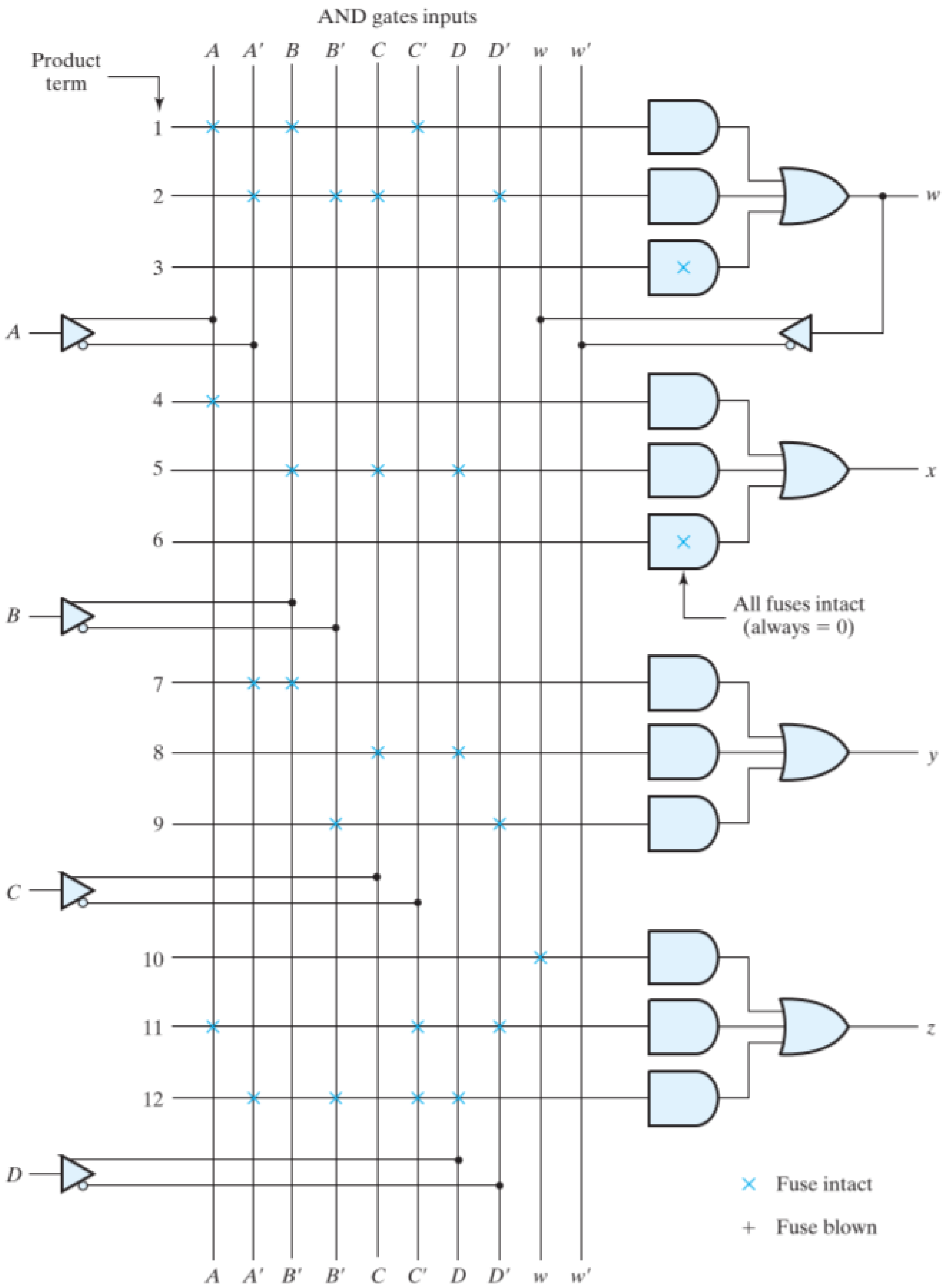
$X = A + BCD$

$Y = A' B + CD + B'D'$

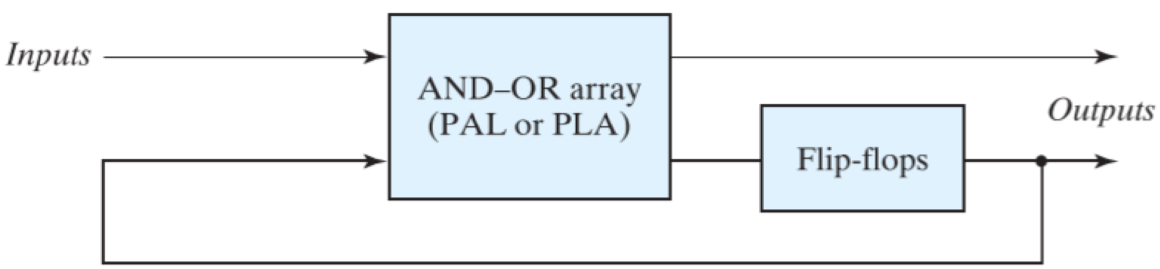
$Z = W + AC'D' + A'B'C'D$

Programming Table

Product Term	**A**	**B**	**C**	**D**	**w**	**Output**
1	1	1	0	_	_	
2	0	0	1	0	_	$w = ABC' + A'B'C D'$
3	_	_	_	_	_	
4	1	_	_	_	_	
5	_	1	1	1	_	$x = A + BCD$
6	_	_	_	_	_	
7	0	1	_	_	_	
8	_	_	1	1	_	$y = A'B + CD + B'D'$
9	_	0	_	0	_	
10	_	_	_	_	1	
11	1	_	0	0	_	$z = w + A C'D' + A'B'C'D$
12	0	0	0	1	_	



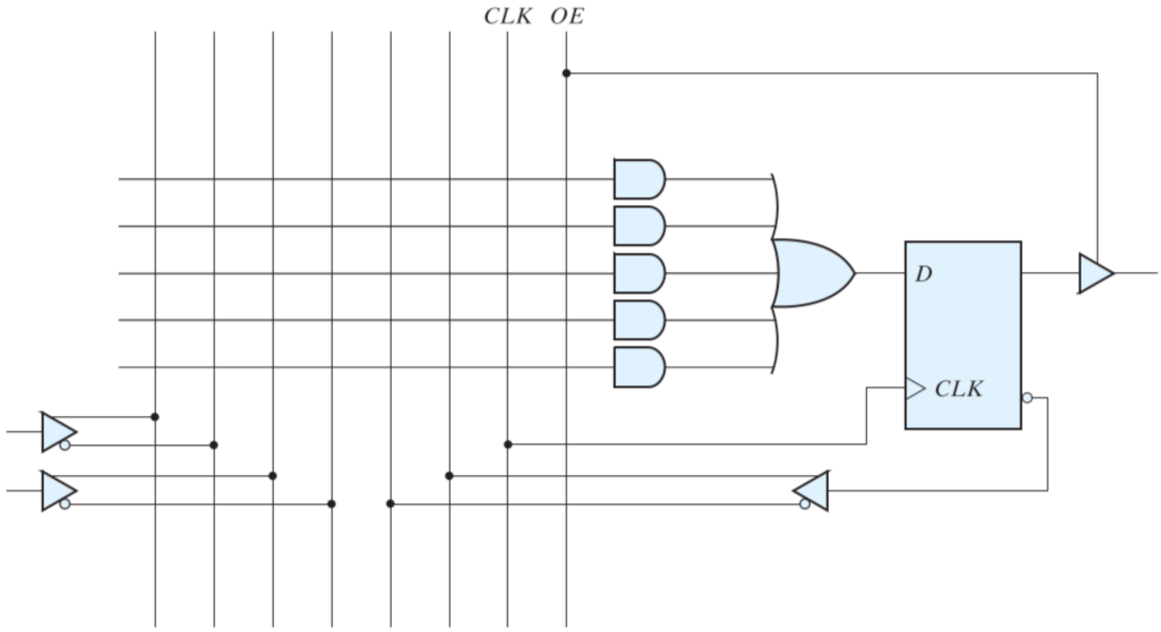
Sequential Programmable Logic Devices



Eg. Field Programmable logic sequencer

- PLA with the output driving a flip-flop

Macro-cell



Output enable (OE): three state buffer

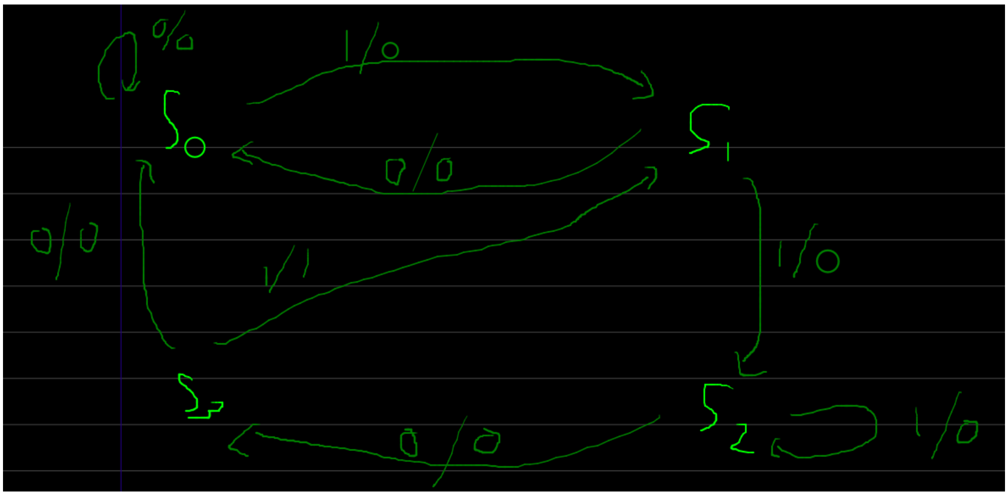
- if OE is 0, output is not obtained
- if OE is 1, output is obtained

Eg. 3-bit up counter which counts when input = 1, remains in same state when input is 0
State Table:

Present State			Next State		
A_2	A_1	A_0	A_2	A_1	A_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

$Q_2^+ = Q_2Q_0' + Q_2Q_1' + Q_2x' + Q_2'Q_1Q_0x$
$Q_1^+ = Q_1Q_0' + Q_1x' + Q_1'Q_0x$
$Q_0^+ = Q_0x' + Q_0'x$

Eg. Design a PLD circuit using PAL/PLA for detecting 1101 Sequence



A	B	x	A+	B+	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

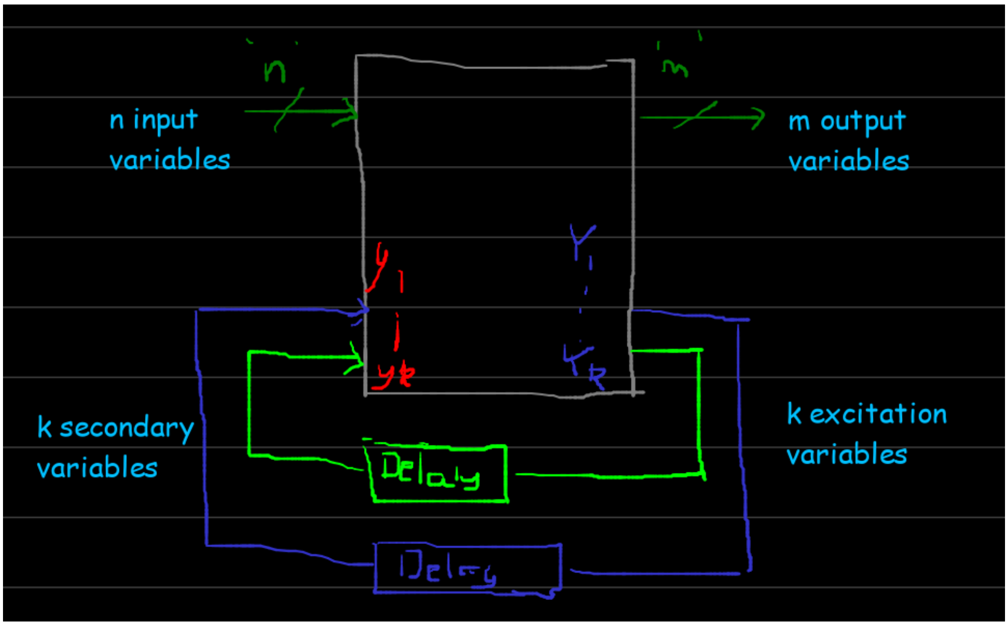
$A_+ = A'Bx + AB'$
$B_+ = A'B'x + AB'x' + ABx$
$y = ABx$

Field programmable gate array (FPGA)

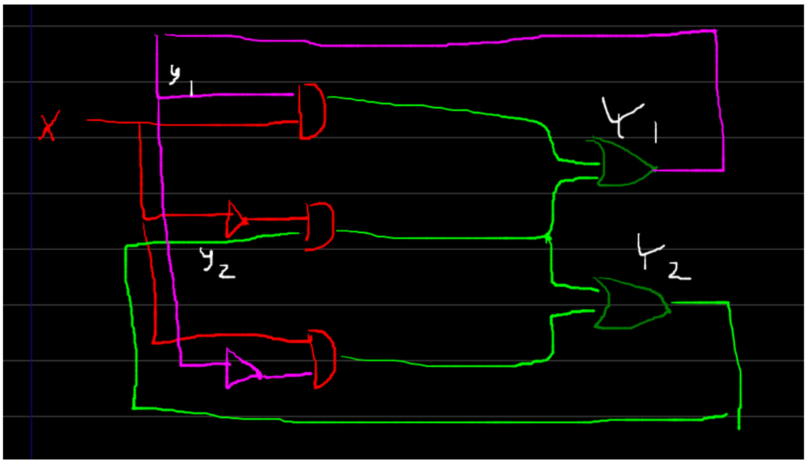
- Reconfigurability
- Rapid prototyping
- Parallel processing (multiple cores)
- Low latency
- Low power consumption→ we're only turning on the gates we want

June 17

Asynchronous sequential circuits



Eg.



x/y_1y_2	**00**	**01**	**11**	**10**
0	0	1	1	0
1	0	0	1	1

x/y_1y_2	**00**	**01**	**11**	**10**
0	0	1	1	0
1	1	1	0	0

Transition Table

x/y_1y_2	**00**	**01**	**11**	**10**
0	00	11	11	00
1	01	01	10	10

Stable states → 00, 11, 01, 10
If any other states are attained, we change to some other state in the next instant

Eg.
Flow Table
Two states with two inputs and one output

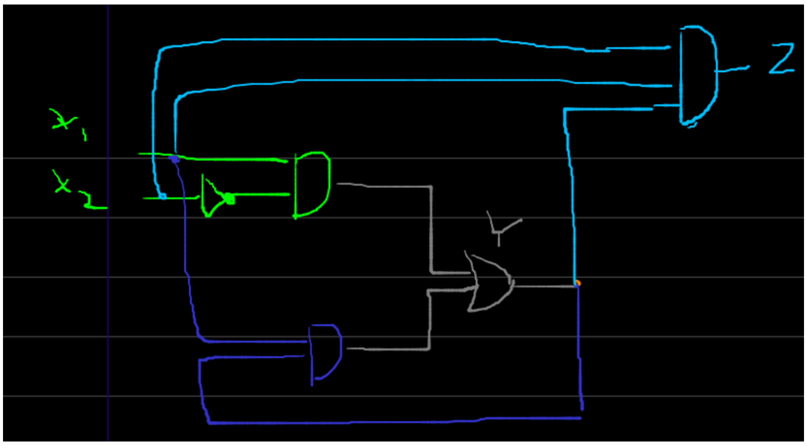
y/x_1x_2	**00**	**01**	**11**	**10**
a	a, 0	a, 0	a, 0	b, 0
b	a, 0	a, 0	b, 1	b, 0

y/x_1x_2	**00**	**01**	**11**	**10**
0	0	0	0	1
1	0	0	1	1

$Y = x_1x_2' + x_1y$

y/x_1x_2	**00**	**01**	**11**	**10**
0	0	0	0	0
1	0	0	1	0

$z = x_1x_2y$



Race conditions
The process of obtaining the logic circuit from the flow table is not always simple
11 → 00 or 01 → 10: might go through some intermediate state, which is fine
Critical Race condition: End up at different state depending on delays
Non-critical Race condition: End up at same state. It can cause hazards though

Eg.

x/y_1y_2	**00**	**01**	**11**	**10**

x/y_1y_2	**00**	**01**	**11**	**10**
0	00			
1	11	11	11	11

00 → 11
00 → 01 → 11
00 → 10 → 11
This is *non-critical race condition*

Eg.

x/y_1y_2	**00**	**01**	**11**	**10**
0	00			
1	11	01	11	10

00 → 11 if delays are equal
00 → 01 or 00 → 10 if delays are unequal
This is *critical race condition*

Eg.

x/y_1y_2	**00**	**01**	**11**	**10**
0	00			
1	11	01	01	11

00 → 01
00 → 10 → 11 → 01
Non-critical race condition

Eg.

x/y_1y_2	**00**	**01**	**11**	**10**
0	00			
1	11	11	11	10

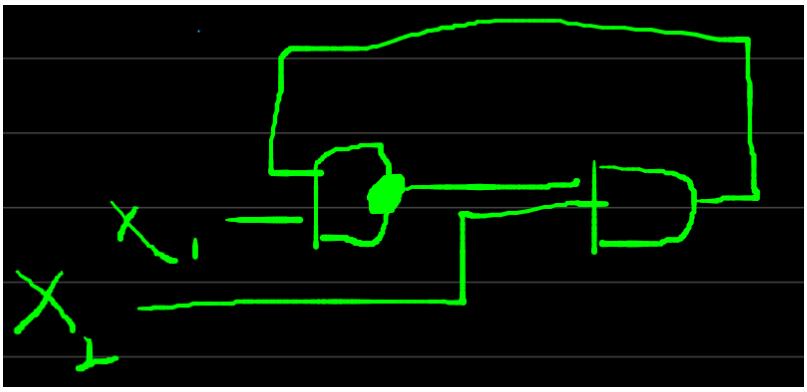
00 → 01 → 11
00 → 10
Critical Race condition

Eg.

x/y_1y_2	**00**	**01**	**11**	**10**
0	00			
1	01	11	10	01

For x = 1, output is unstable and cycles as follows: *00 → 01 → 11 → 10 → 01 → 11 → 10...*

Determining Stability



$Y = (x_1y)'x_2$

Transition Table:

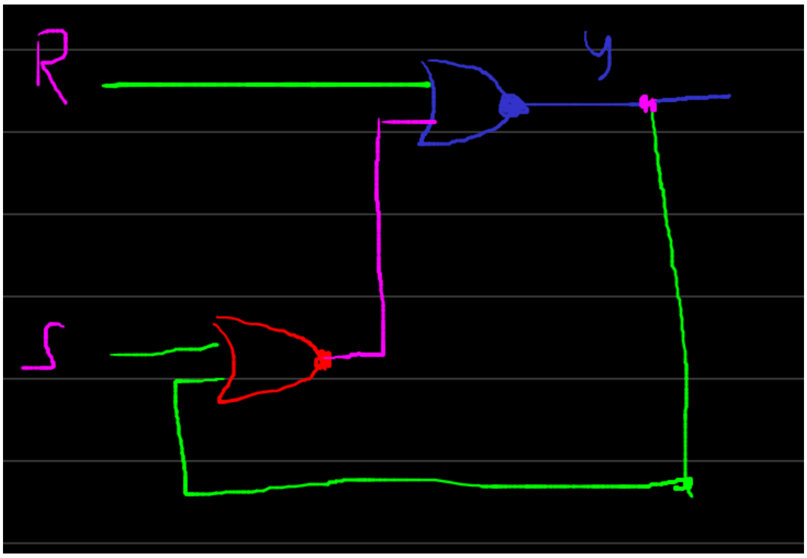
y/x_1x_2	**00**	**01**	**11**	**10**
0	0	1	1	0
1	0	1	0	0

Unstable circuit, as for input 11, output switches rapidly between two states

Analysis

- Determine all feedback loops
- Derive Boolean function for each y_i
- Plot each y in a map
- Combine all maps (transition table)
- Identify and circle stable states
- Identify race conditions

SR latch:
$Y = ((S + y)' + R)' = (S + y)R' = R'S + R'y$



y / SR	00	01	11	10
0	0	0	0	1
1	1	0	0	1

- It is stable
- Race conditions checking
  - 00 → 11
  - 00 → 01 → 11
  0 0 0
  - 00 → 10 → 11 → 11
  0 1 0 0
  - 00 → 01 → 11
  1 0 0
  - 00 → 10 → 11 → 11
  1 1 0 0
  - 11 → 00
  - 11 → 10 → 00
  0 1 1
  - 11 → 01 → 00
  0 0 0
  - 11 → 10 → 00
  1 Unstable
  - 11 → 01 → 11
  1 Unstable
  - 01 → 10
  - 01 → 00 → 10 → 10
  0 0 1 1
  - 01 → 11 → 10 → 10
  0 0 1 1
  - 01 → 00 → 10

1 Unstable
- 01 → 00 → 10

1 Unstable
- 10 → 01
- 10 → 00 → 01

0 Unstable
- 10 → 11 → 01

0 Unstable
- 10 → 00 → 01 → 01

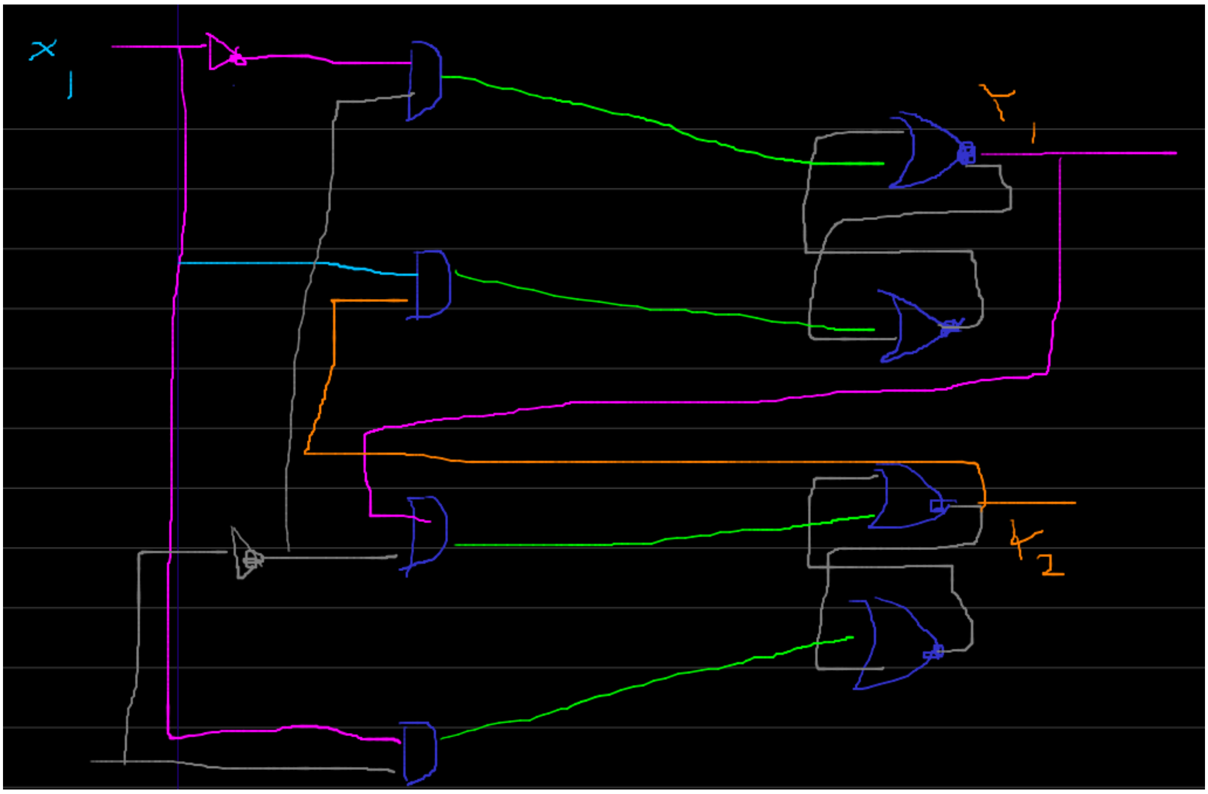
1 1 0 0
, - 10 → 11 → 01
, 1 0

11 → 00
S → 0 first y → 0
R → 0 first y → 1

- we have to ensure $SR = 0$ to prevent 11 state

$Y = ((S+y)' + R)' = (S+y)R' = R'S + R'y + 0 = R'S + R'y + SR = S + R'y$

Finding Critical Race conditions
Eg.



Transition Table:

y_1y_2/x_1x_2	**00**	**01**	**11**	**10**

y_1y_2/x_1x_2	**00**	**01**	**11**	**10**
00	00	00	01	00
01	01	01	11	11
11	00	11	11	10
10	00	10	11	10

For each input, at least one stable state → stable

Race condition:

- Input 00
  No 00 → 11, 11 → 00, 01 → 10, 10 → 01 conditions
- Input 01, state 11
  11 → 00: can be done when input is 00
  But, y_1y_2 might go as follows:
- 11 → 10 → 00
- 11 → 01
  Critical Race condition
- Input 01, no other conditions possible
- Input 11, state 11
  11 → 00, input given must be 00
- 11 → 01 → 01
- 11 → 10 → 00
  Critical Race condition
- Input 10, no other conditions possible

Implementation of circuit using SR latch
Eg. $Y = x_1x_2' + x_1y$

y/x_1x_2	**00**	**01**	**11**	**10**
0	0	0	0	1
1	0	0	1	1

Excitation Table

y	Y	S	R

y	Y	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

S

y/x_1x_2	**00**	**01**	**11**	**10**
0	0	0	0	1
1	0	0	X	X

S = x_1x_2'

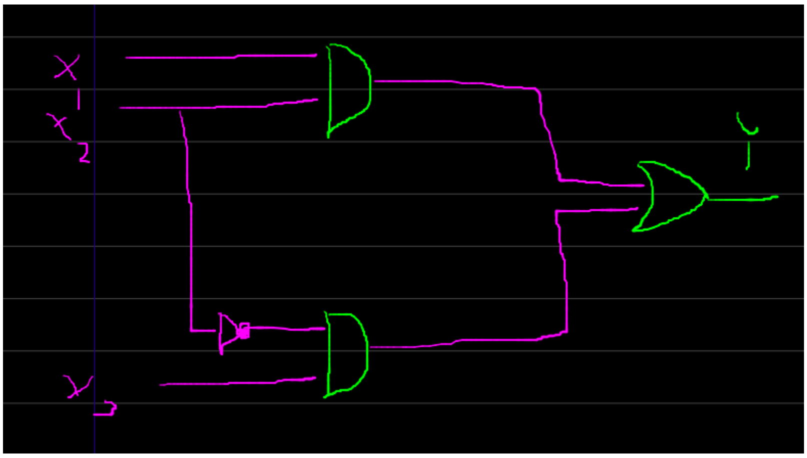
R

y/x_1x_2	**00**	**01**	**11**	**10**
0	X	X	X	0
1	1	1	0	0

R = x_1'

Hazards → Malfunction/switching transient due to unequal delays

Static 1 hazard



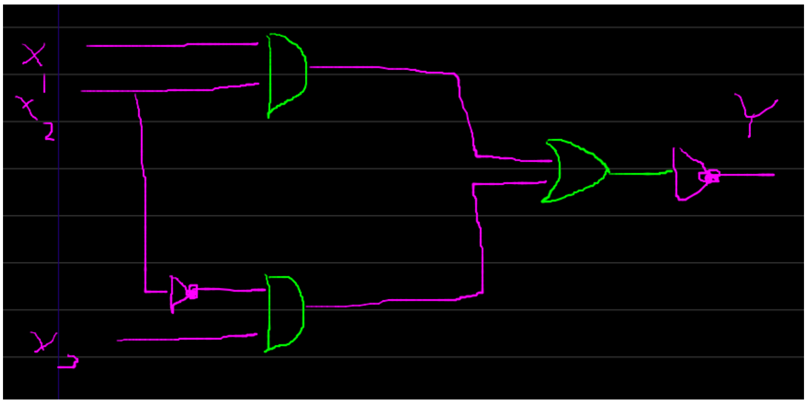
$x_1 = 1, x_3 = 1, x_2 : 1 \to 0$
Y becomes 0 for an instant, then becomes 1 again

The function is $Y = x_1x_2 + x_3x_2'$

x_1/x_2x_3	**00**	**01**	**11**	**10**
0	0	1	0	0
1	0	1	1	1

Adjacent 1s without common implicant is static 1 hazard

Static 0 hazard



Adjacent 0s without common implicant is static 0 hazard