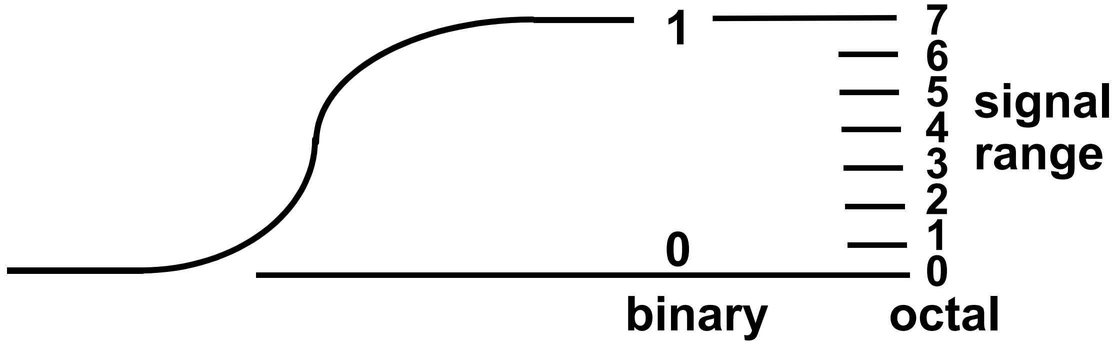
DIGITAL LOGIC CIRCUITS

LOGIC GATES

Digital Computers

- Imply that the computer deals with digital information, i.e., it deals with the information that is represented by binary digits
- Why *BINARY* ? instead of Decimal or other number system ?

* Consider electronic signal

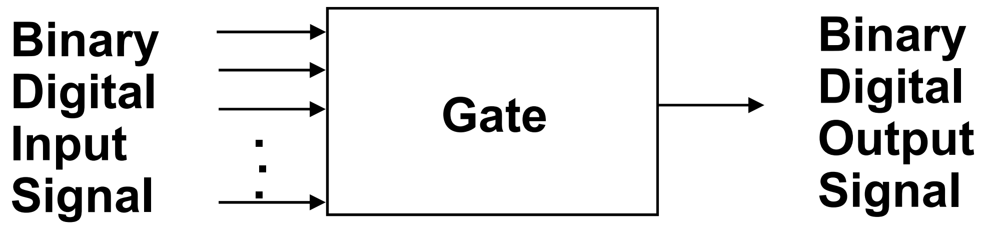


* Consider the calculation cost - Add

	0	1
0	0	1
1	1	10

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

BASIC LOGIC BLOCK - GATE -



Types of Basic Logic Blocks

- **Combinational Logic Block**
  Logic Blocks whose output logic value depends only on the input logic values

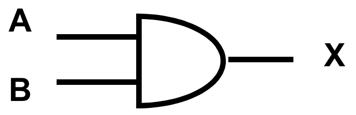
- **Sequential Logic Block**
  Logic Blocks whose output logic value depends on the input values and the state (stored information) of the blocks

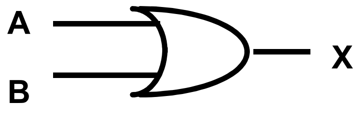
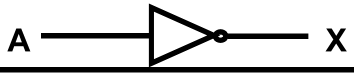
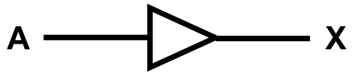
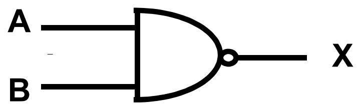
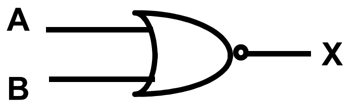
Functions of Gates can be described by

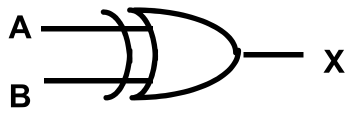
- **Truth Table**
- **Boolean Function**
- **Karnaugh Map**

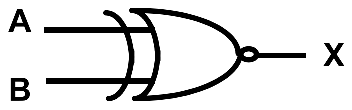
COMBINATIONAL GATES

Name	Symbol	Function	Truth Table

AND		X = A • B
		or
		X = AB
			A B | X
			0 0 | 0
			0 1 | 0
			1 0 | 0
			1 1 | 1

OR		X = A + B	A B | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
I		X = A’	A | X 0 | 1 1 | 0
Buffer		X = A	A | X 0 | 0 1 | 1
NAND		X = (AB)’	A B | X 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		X = (A + B)’	A B | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

XOR
Exclusive OR
		X = A ⊕ B
		or
		X = A’B + AB’
			A B | X
			0 0 | 0
			0 1 | 1
			1 0 | 1
			1 1 | 0

XNOR
Exclusive NOR
or Equivalence
		X = (A ⊕ B)’
		or
		X = A’B’+ AB
			A B | X
			0 0 | 1
			0 1 | 0
			1 0 | 0
			1 1 | 1

BOOLEAN ALGEBRA

Boolean Algebra

* Algebra with Binary(Boolean) Variable and Logic Operations
* Boolean Algebra is useful in Analysis and Synthesis of Digital Logic Circuits

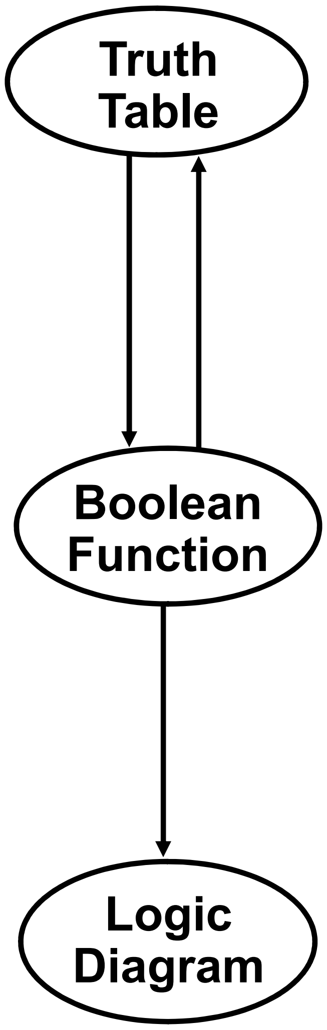
- Input and Output signals can be represented by Boolean Variables, and
- Function of the Digital Logic Circuits can be represented by Logic Operations, i.e., Boolean Function(s)
- From a Boolean function, a logic diagram can be constructed using AND, OR, and I

Truth Table

* The most elementary specification of the function of a Digital Logic Circuit is the Truth Table

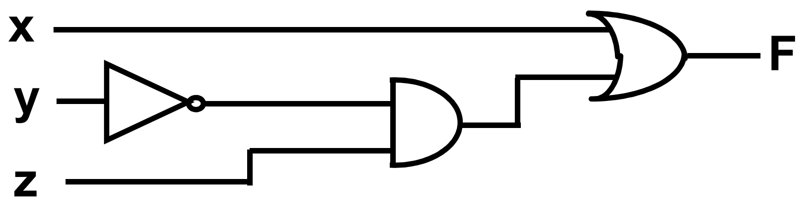
- Table that describes the Output Values for all the combinations of the Input Values, called *MINTERMS*
- n input variables $\rightarrow 2^n$ minterms

LOGIC CIRCUIT DESIGN



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

F = x + y'z



BASIC IDENTITIES OF BOOLEAN ALGEBRA

[1] $x + 0 = x$	[2] $x \bullet 0 = 0$
[3] $x + 1 = 1$	[4] $x \bullet 1 = x$
[5] $x + x = x$	[6] $x \bullet x = x$
[7] $x + x' = 1$	[8] $x \bullet X' = 0$
[9] $x + y = y + x$	[10] $xy = yx$
[11] $x + (y + z) = (x + y) + z$	[12] $x(yz) = (xy)z$
[13] $x(y + z) = xy + xz$	[14] $x + yz = (x + y)(x + z)$
[15] $(x + y)' = x'y'$	[16] $(xy)' = x' + y'$
[17] $(x')' = x$	

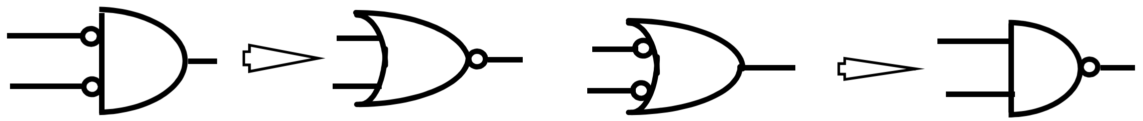
[15] and [16] : De Morgan's Theorem

Usefulness of this Table

- Simplification of the Boolean function
- Derivation of equivalent Boolean functions to obtain logic diagrams utilizing different logic gates
  -- Ordinarily ANDs, ORs, and Inverters
  -- But a certain different form of Boolean function may be convenient to obtain circuits with NANDs or NORs
  $\rightarrow$ Applications of De Morgans Theorem

$x'y' = (x + y)'$ I, AND $\rightarrow$ NOR

$x' + y' = (xy)'$ I, OR $\rightarrow$ NAND

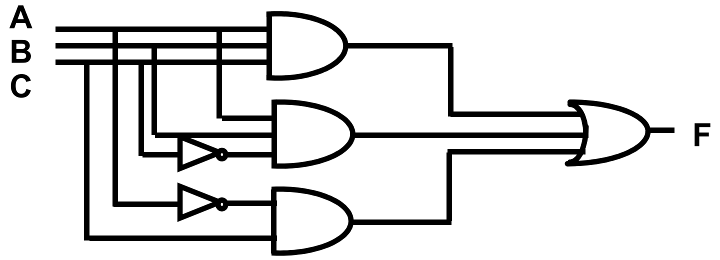


EQUIVALENT CIRCUITS

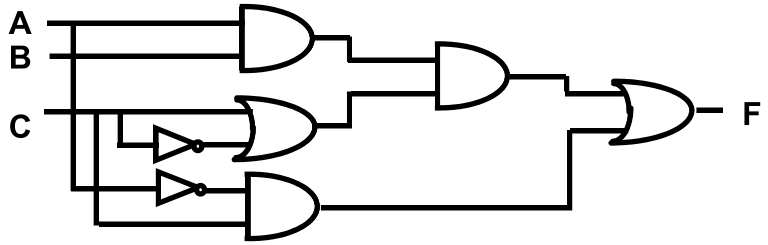
Many different logic diagrams are possible for a given Function

$$
\begin{aligned}
F &= ABC + ABC' + A'C && \text{.............} \quad (1)\\
&= AB(C + C') + A'C && [13] \ \text{......} \quad (2)\\
&= AB \bullet 1 + A'C && [7]\\
&= AB + A'C && [4] \ \text{.......} \quad (3)
\end{aligned}
$$

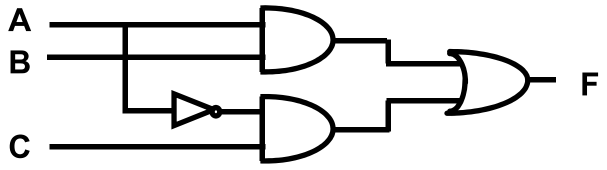
(1)



(2)



(3)



COMPLEMENT OF FUNCTIONS

A Boolean function of a digital logic circuit is represented by only using logical variables and AND, OR, and Invert operators.

→ Complement of a Boolean function

- Replace all the variables and subexpressions in the parentheses appearing in the function expression with their respective complements

$$A,B,\ldots,Z,a,b,\ldots,z \Rightarrow A',B',\ldots,Z',a',b',\ldots,z'$$
$$(p + q) \Rightarrow (p + q)'$$

- Replace all the operators with their respective complementary operators

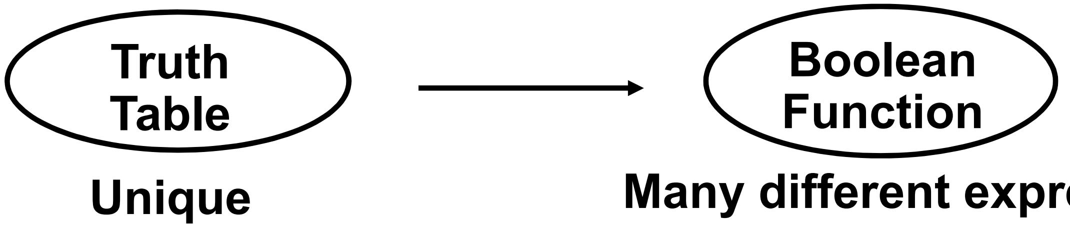
$$\text{AND} \Rightarrow \text{OR}$$
$$\text{OR} \Rightarrow \text{AND}$$

- Basically, extensive applications of the De Morgan's theorem

$$(x_1 + x_2 + \ldots + x_n)' \Rightarrow x_1'x_2'\ldots x_n'$$

$$(x_1x_2 \ldots x_n)' \Rightarrow x_1' + x_2' + \ldots + x_n'$$

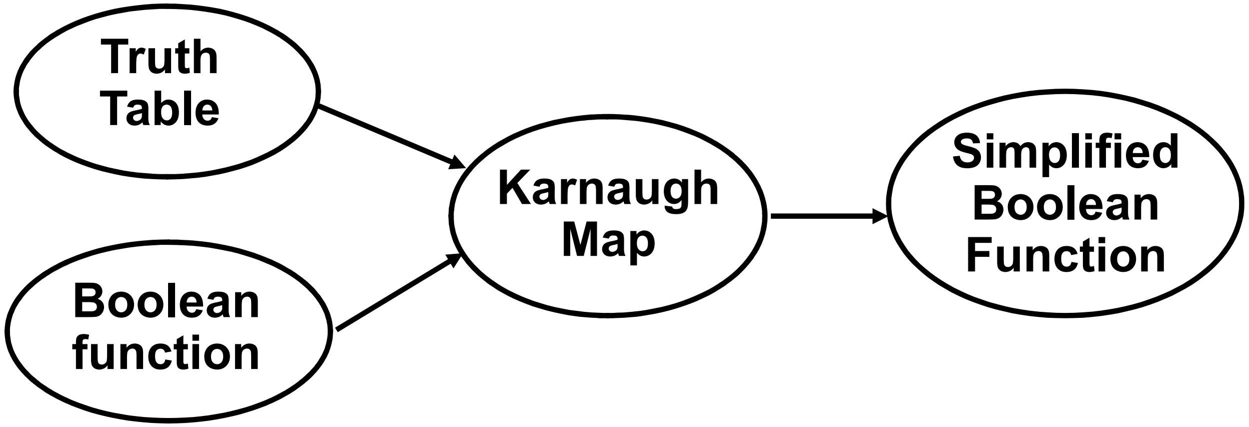
SIMPLIFICATION



Simplification from Boolean function

- Finding an equivalent expression that is least expensive to implement
- For a simple function, it is possible to obtain a simple expression for low cost implementation
- But, with complex functions, it is a very difficult task

Karnaugh Map (K-map) is a simple procedure for simplifying Boolean expressions.



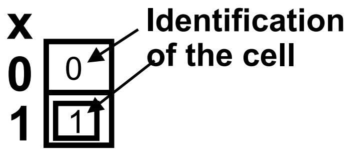
KARNAUGH MAP

Karnaugh Map for an n-input digital logic circuit (n-variable sum-of-products form of Boolean Function, or Truth Table) is

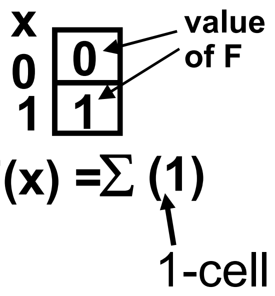
- **Rectangle divided into 2^n cells**
- **Each cell is associated with a *Minterm***
- **An output(function) value for each input value associated with a mintern is written in the cell representing the minterm**
  → 1-cell, 0-cell

Each Minterm is identified by a decimal number whose binary representation is identical to the binary interpretation of the input values of the minterm.

x	F
0	1
1	0

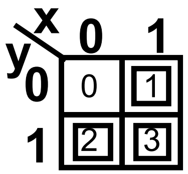


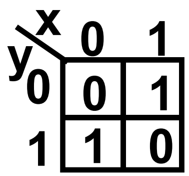
Karnaugh Map



F(x) = $\sum$ (1)

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

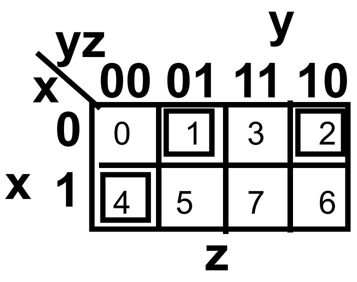




F(x,y) = $\sum$ (1,2)

KARNAUGH MAP

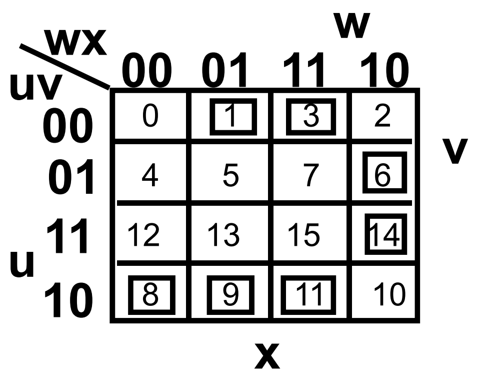
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0



x \ yz	00	01	11	10
0	0	1	0	1
1	1	0	0	0

F(x,y,z) = $\sum$ (1,2,4)

u	v	w	x	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0



uv \ wx	00	01	11	10
00	0	1	1	0
01	0	0	0	1
11	0	0	0	1
10	1	1	1	0

F(u,v,w,x) = $\sum$ (1,3,6,8,9,11,14)

MAP SIMPLIFICATION - 2 ADJACENT CELLS -

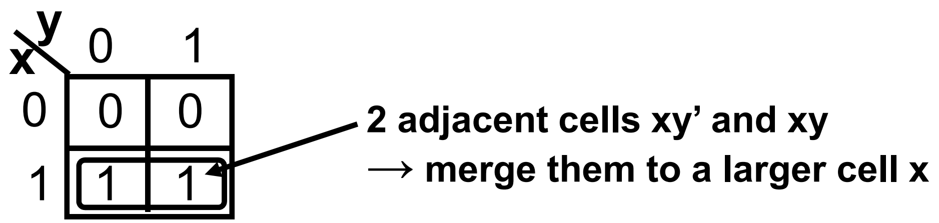
Rule: $xy' + xy = x(y+y') = x$

Adjacent cells

- binary identifications are different in one bit
→ minterms associated with the adjacent cells have one variable complemented each other

Cells (1,0) and (1,1) are adjacent
Minterms for (1,0) and (1,1) are
$x \cdot y'$ --> x=1, y=0
$x \cdot y$ --> x=1, y=1

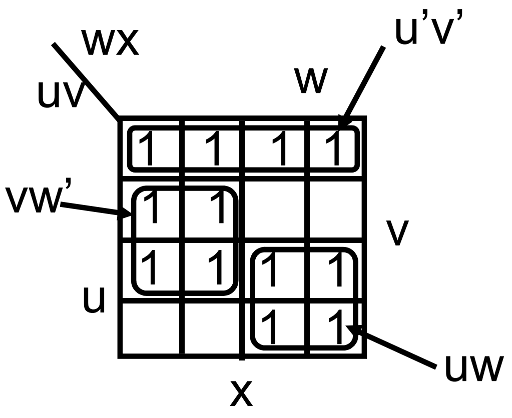
F = xy'+ xy can be reduced to F = x
From the map

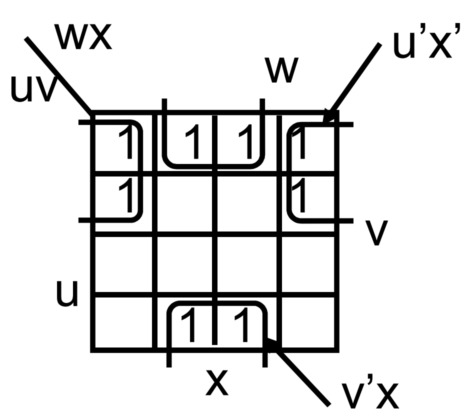


$$F(x,y) = \sum(2,3)$$
$$= xy' + xy$$
$$= x$$

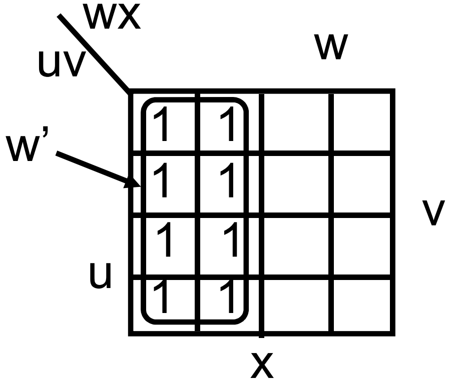
MAP SIMPLIFICATION - MORE THAN 2 CELLS -

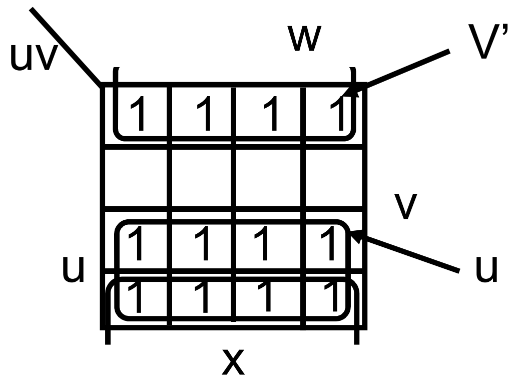
u’v’w’x’ + u’v’w’x + u’v’wx + u’v’wx’
= u’v’w’(x’+x) + u’v’w(x+x’)
= u’v’w’ + u’v’w
= u’v’(w’+w)
= u’v’





u’v’w’x’+u’v’w’x+u’vw’x’+u’vw’x+uvw’x’+uvw’x+uv’w’x’+uv’w’x
= u’v’w’(x’+x) + u’vw’(x’+x) + uvw’(x’+x) + uv’w’(x’+x)
= u’(v’+v)w’ + u(v’+v)w’
= (u’+u)w’ = w’

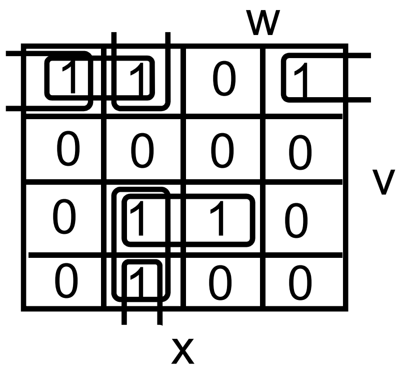




MAP SIMPLIFICATION

uv \ wx	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	0	1	1	0
10	0	1	0	0

$\longrightarrow$



$F(u,v,w,x) = \sum (0,1,2,9,13,15)$

Adjacent Cells of 1
(0,1), (0,2), (0,4), (0,8)
Adjacent Cells of 0
(1,0), (1,3), (1,5), (1,9)
...
...
Adjacent Cells of 15
(15,7), (15,11), (15,13), (15,14)

$\longrightarrow$

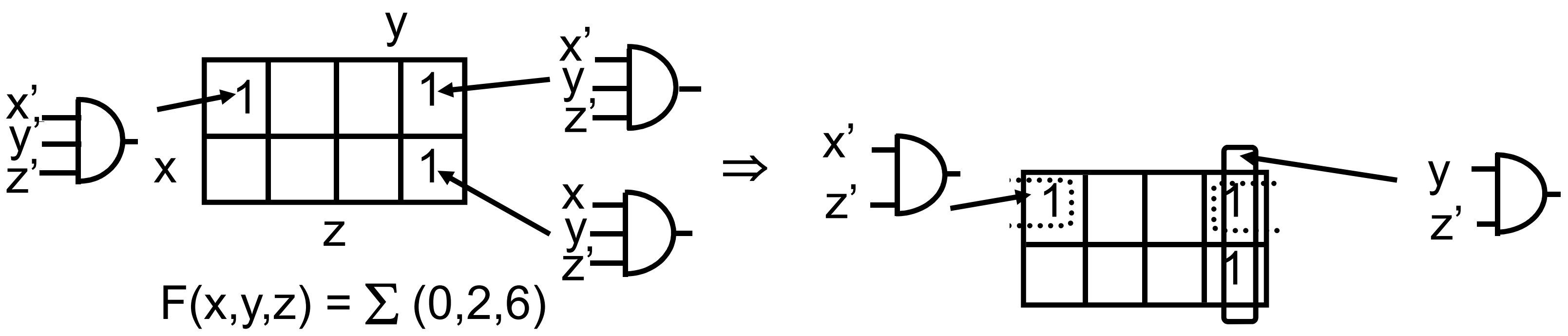
Merge (0,1) and (0,2)
--> u'v'w' + u'v'x'
Merge (1,9)
--> v'w'x
Merge (9,13)
--> uw'x
Merge (13,15)
--> uvx

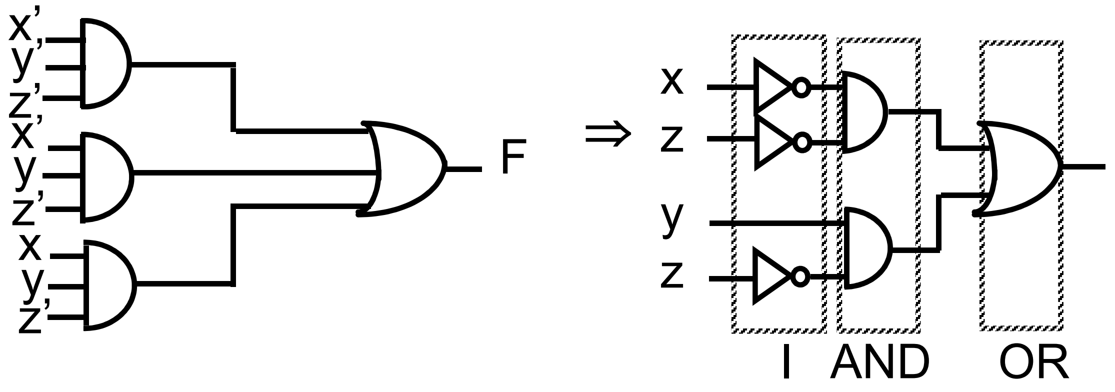
F = u'v'w' + u'v'x' + v'w'x + uw'x + uvx
But (9,13) is covered by (1,9) and (13,15)
F = u'v'w' + u'v'x' + v'w'x + uvx

IMPLEMENTATION OF K-MAPS - Sum-of-Products Form -

Logic function represented by a Karnaugh map can be implemented in the form of I-AND-OR

A cell or a collection of the adjacent 1-cells can be realized by an AND gate, with some inversion of the input variables.



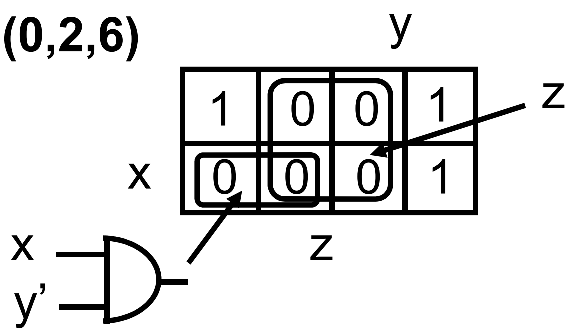


IMPLEMENTATION OF K-MAPS - Product-of-Sums Form -

Logic function represented by a Karnaugh map can be implemented in the form of I-OR-AND

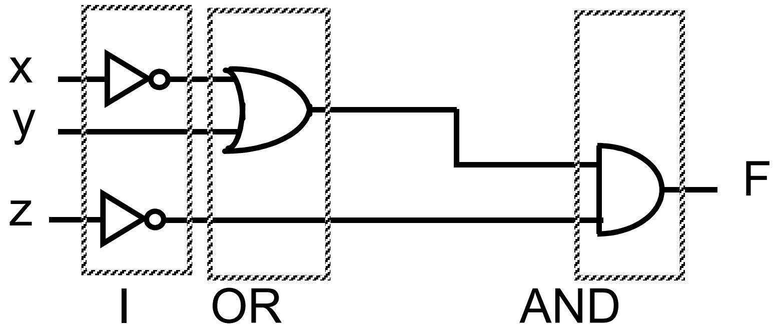
If we implement a Karnaugh map using 0-cells, the complement of F, i.e., F', can be obtained. Thus, by complementing F' using DeMorgan's theorem F can be obtained

F(x,y,z) = (0,2,6)



$F' = xy' + z$

$F = (xy')z'$
$= (x' + y)z'$

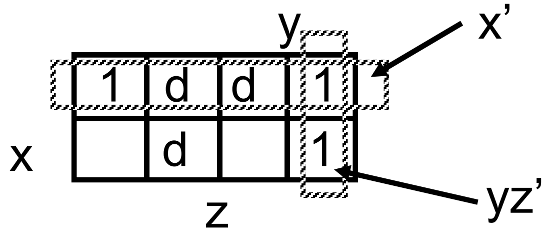


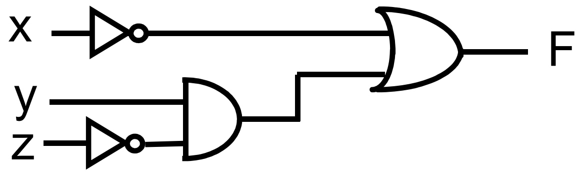
IMPLEMENTATION OF K-MAPS
- Don't-Care Conditions -

In some logic circuits, the output responses for some input conditions are don't care whether they are 1 or 0.

In K-maps, don't-care conditions are represented by d's in the corresponding cells.

Don't-care conditions are useful in minimizing the logic functions using K-map.
- **Can be considered either 1 or 0**
- **Thus increases the chances of merging cells into the larger cells**
  --> Reduce the number of variables in the product terms





COMBINATIONAL LOGIC CIRCUITS

Half Adder

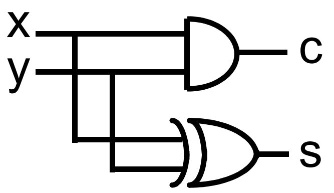
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{array}{c|cc} & & y \\ \hline & 0 & 0 \\ x & 0 & 1 \end{array}$$

$c = xy$

$$\begin{array}{c|cc} & & y \\ \hline & 0 & 1 \\ x & 1 & 0 \end{array}$$

$s = xy' + x'y$
$\quad = x \oplus y$



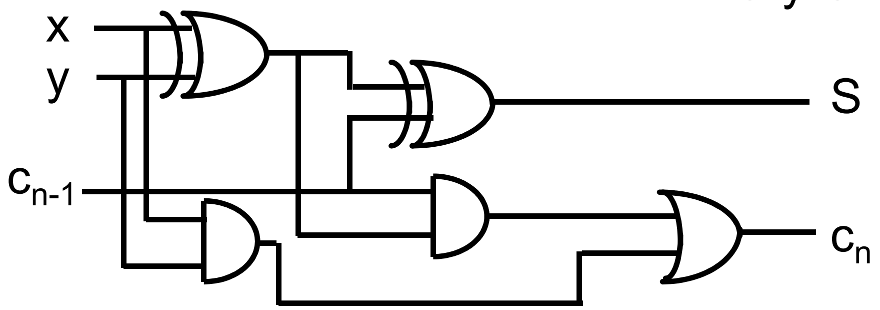
Full Adder

x	y	c_{n-1}	c_n	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

$$\begin{array}{c|cc|c} & & y & \\ \hline & 0 & 0 & \\ & 0 & \boxed{1} & c_{n-1} \\ x & \boxed{1} & \boxed{1} & c_{n-1} \\ x & 0 & \boxed{1} & \\ \hline & c_n & & \end{array}$$

$$\begin{array}{c|cc|c} & & y & \\ \hline & 0 & \boxed{1} & \\ & \boxed{1} & 0 & c_{n-1} \\ x & 0 & \boxed{1} & c_{n-1} \\ x & \boxed{1} & 0 & \\ \hline & s & & \end{array}$$

$c_n = xy + xc_{n-1} + yc_{n-1}$
$\quad = xy + (x \oplus y)c_{n-1}$
$s = x'y'c_{n-1} + x'yc'_{n-1} + xy'c'_{n-1} + xyc_{n-1}$
$\quad = x \oplus y \oplus c_{n-1} = (x \oplus y) \oplus c_{n-1}$



COMBINATIONAL LOGIC CIRCUITS

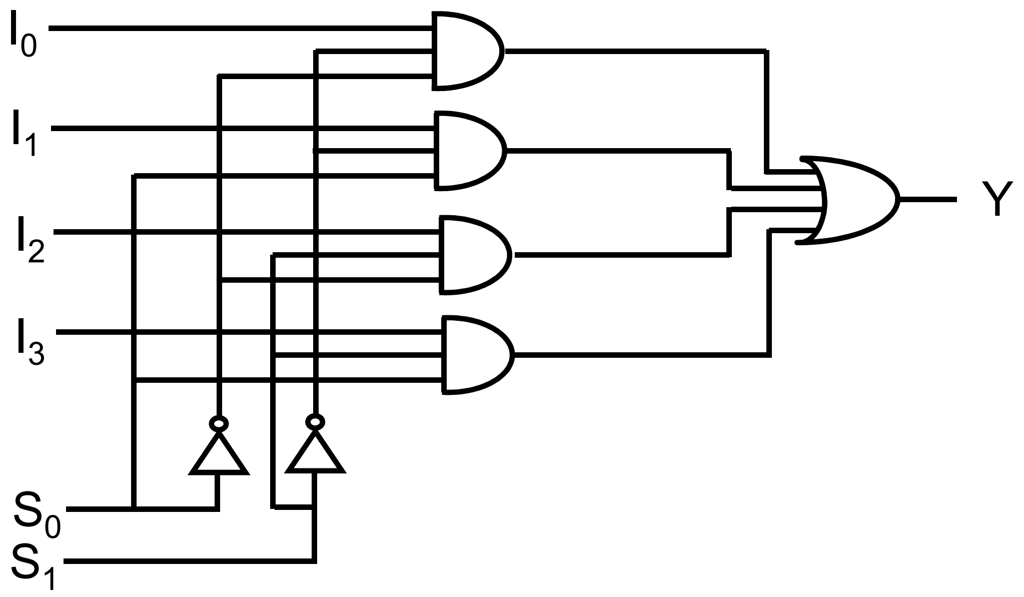
Other Combinational Circuits

Multiplexer
Encoder
Decoder
Parity Checker
Parity Generator
etc

MULTIPLEXER

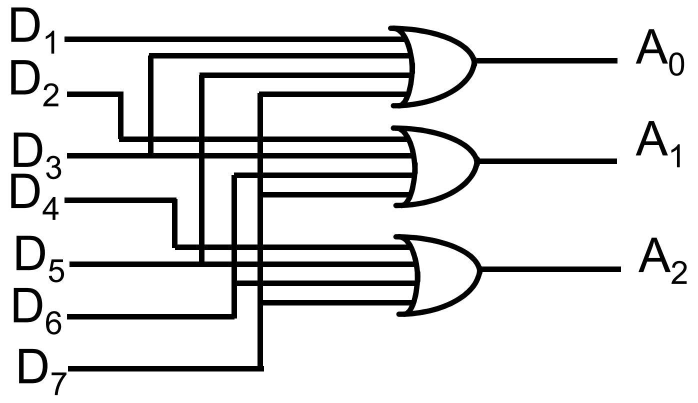
4-to-1 Multiplexer

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



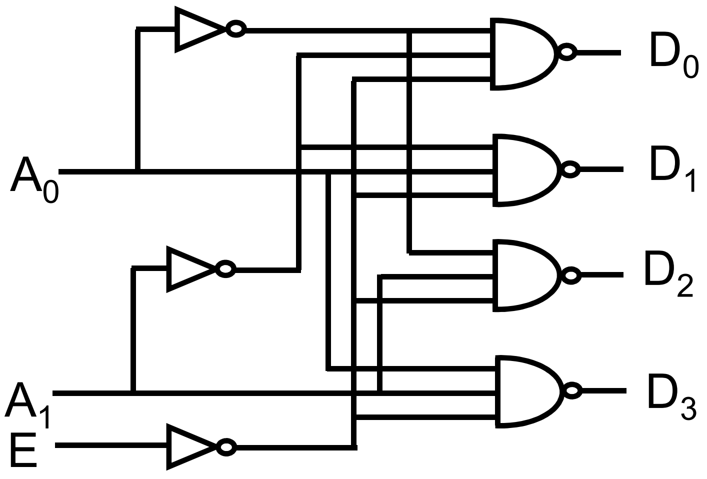
ENCODER/DECODER

Octal-to-Binary Encoder



2-to-4 Decoder

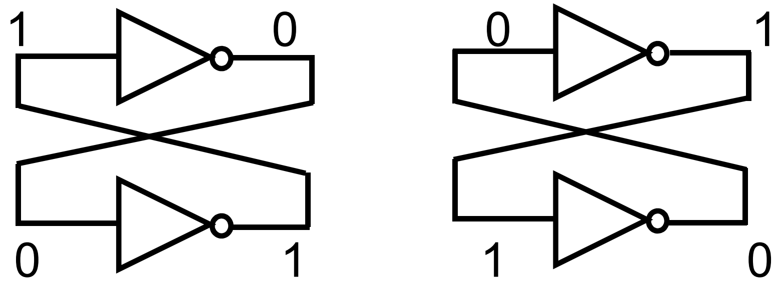
E	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	d	d	1	1	1	1



FLIP FLOPS

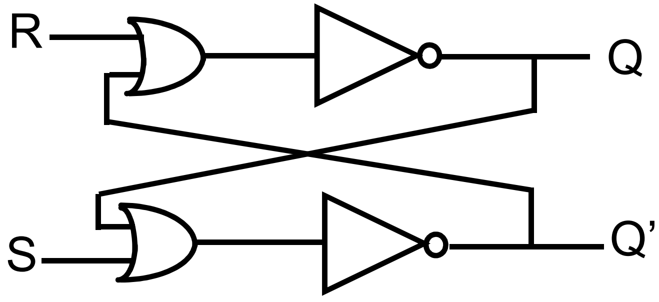
Characteristics

- **2 stable states**
- **Memory capability**
- **Operation is specified by a Characteristic Table**



0-state **1-state**

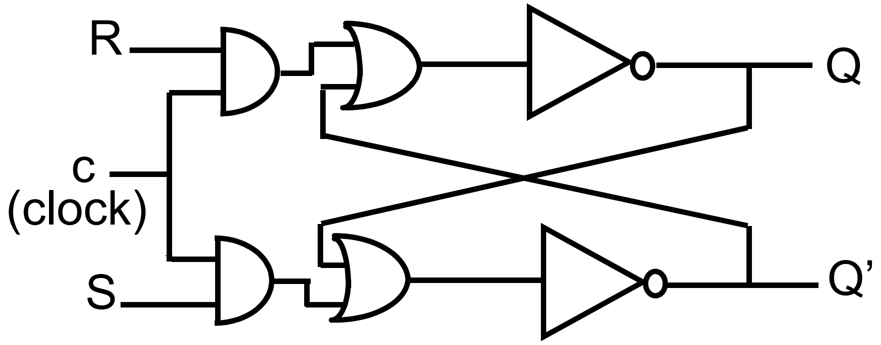
In order to be used in the computer circuits, state of the flip flop should have input terminals and output terminals so that it can be set to a certain state, and its state can be read externally.



S	R	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	indeterminate (forbidden)

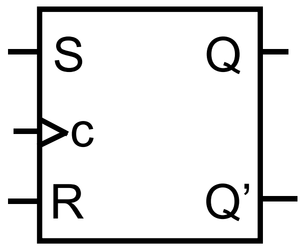
CLOCKED FLIP FLOPS

In a large digital system with many flip flops, operations of individual flip flops are required to be synchronized to a clock pulse. Otherwise, the operations of the system may be unpredictable.

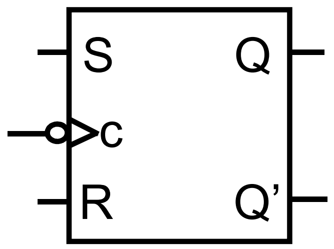


Clock pulse allows the flip flop to change state only when there is a clock pulse appearing at the c terminal.

We call above flip flop a Clocked RS Latch, and symbolically as

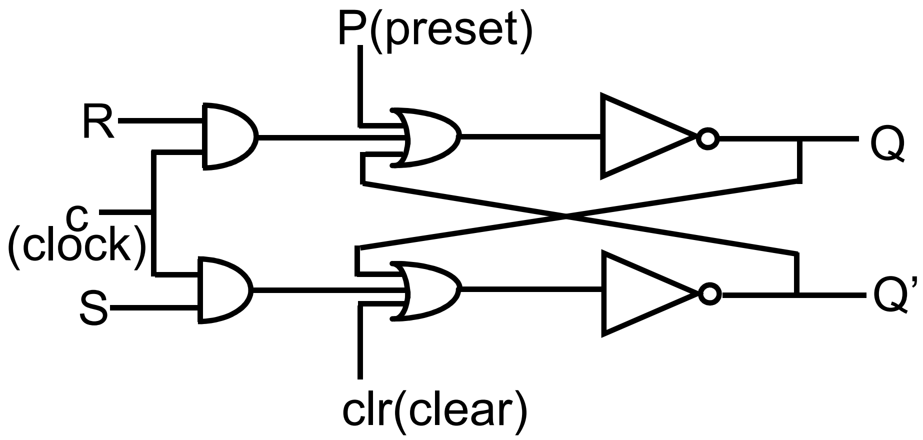


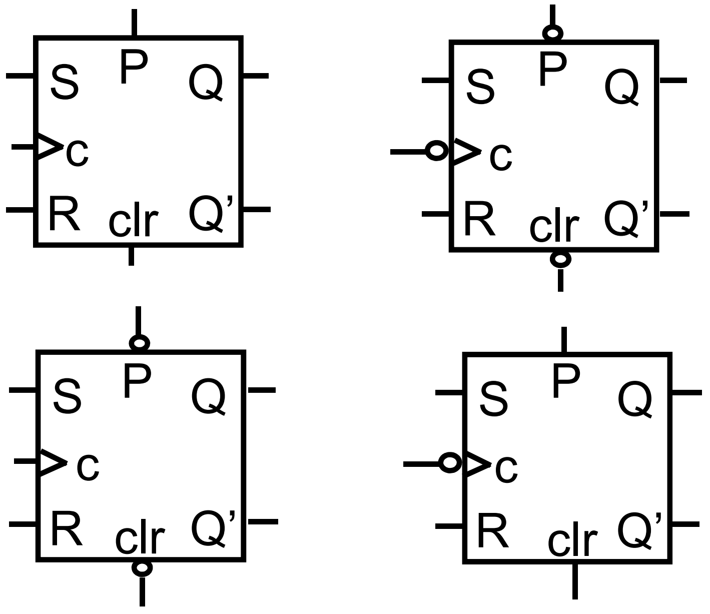
operates when clock is high



operates when clock is low

RS-LATCH WITH PRESET AND CLEAR INPUTS

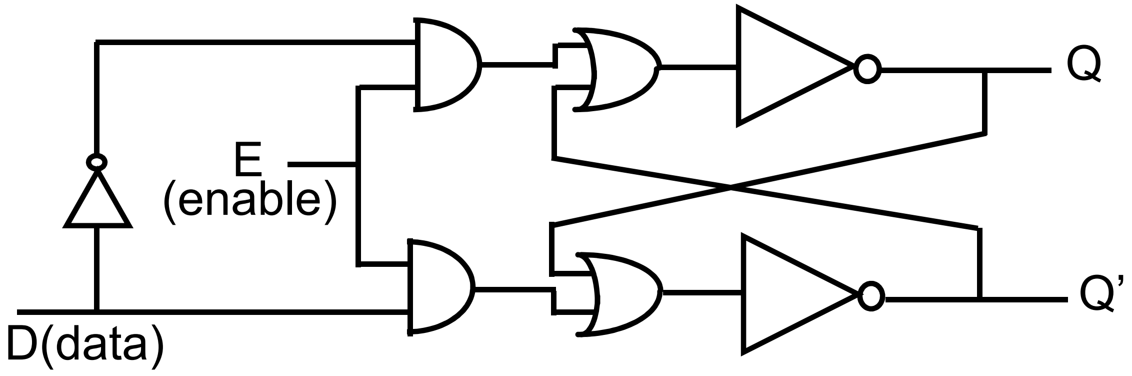


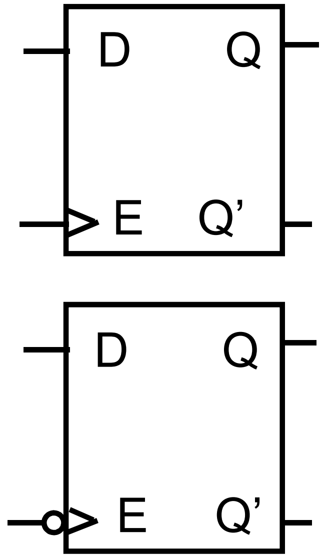


D-LATCH

D-Latch

Forbidden input values are forced not to occur by using an inverter between the inputs





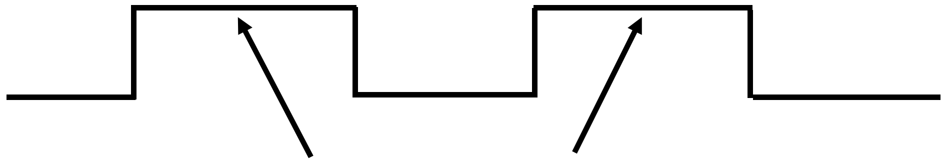
D	Q(t+1)
0	0
1	1

EDGE-TRIGGERED FLIP FLOPS

Characteristics

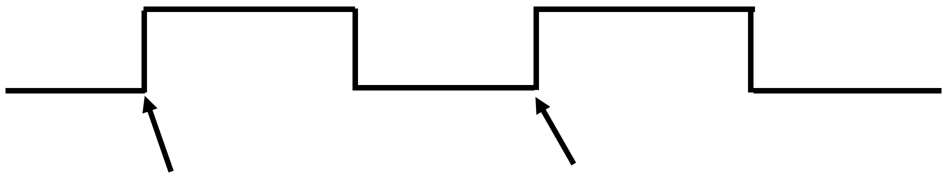
- State transition occurs at the rising edge or falling edge of the clock pulse

Latches



respond to the input only during these periods

Edge-triggered Flip Flops (positive)

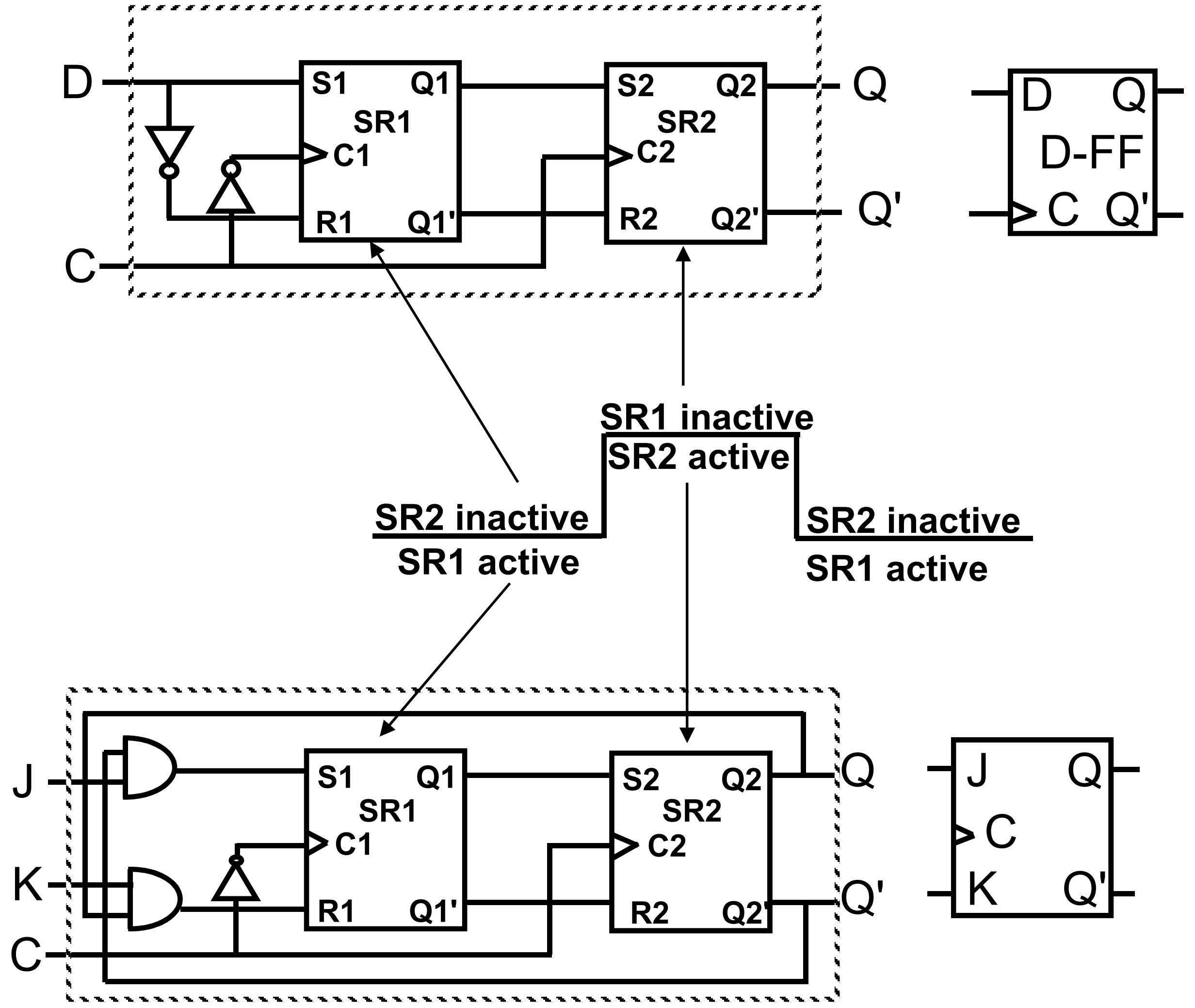


respond to the input only at this time

POSITIVE EDGE-TRIGGERED

D-Flip Flop

JK-Flip Flop

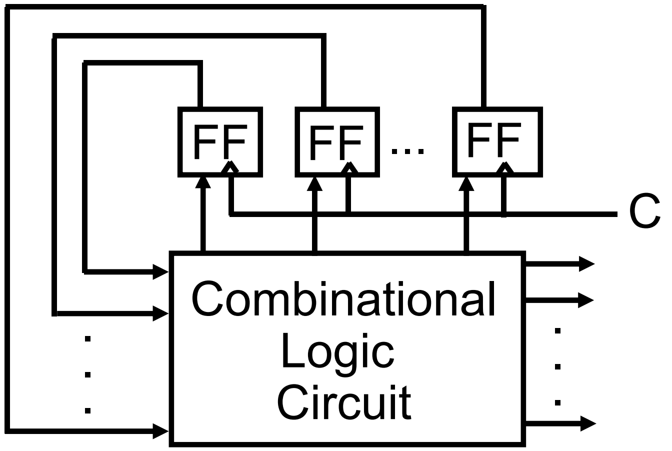


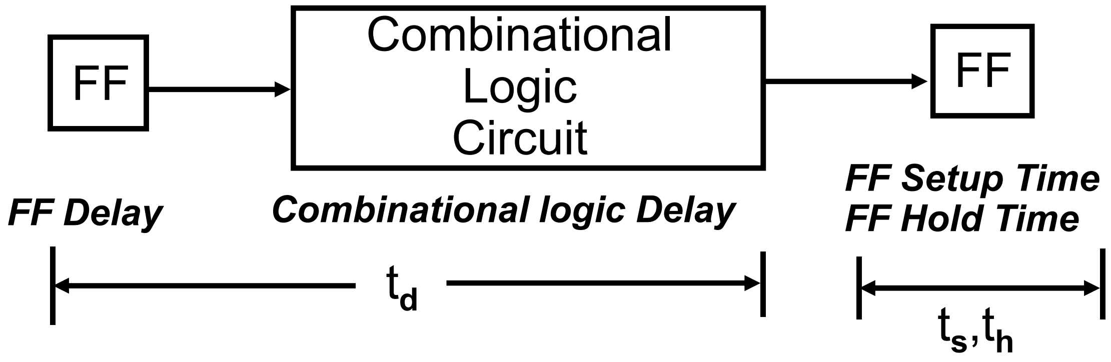
T-Flip Flop: JK-Flip Flop whose J and K inputs are tied together to make T input. Toggles whenever there is a pulse on T input.

CLOCK PERIOD

Clock period determines how fast the digital circuit operates.
How can we determine the clock period ?

Usually, digital circuits are sequential circuits which has some flip flops





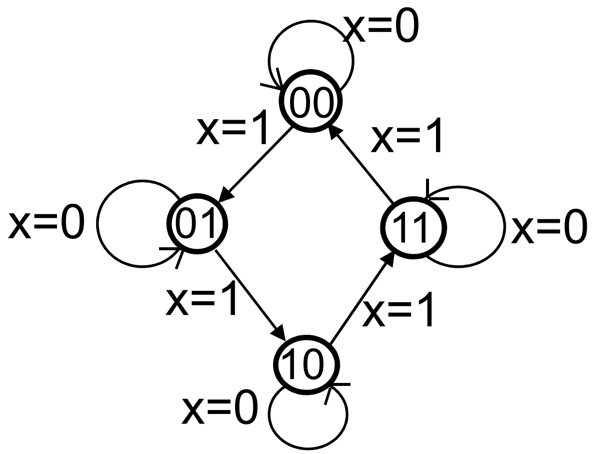
clock period $T = t_d + t_s + t_h$

DESIGN EXAMPLE

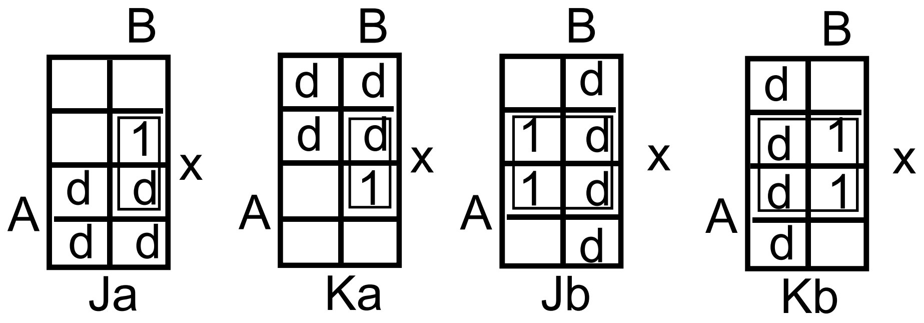
Design Procedure:

Specification ⇒ State Diagram ⇒ State Table ⇒ Excitation Table ⇒ Karnaugh Map ⇒ Circuit Diagram

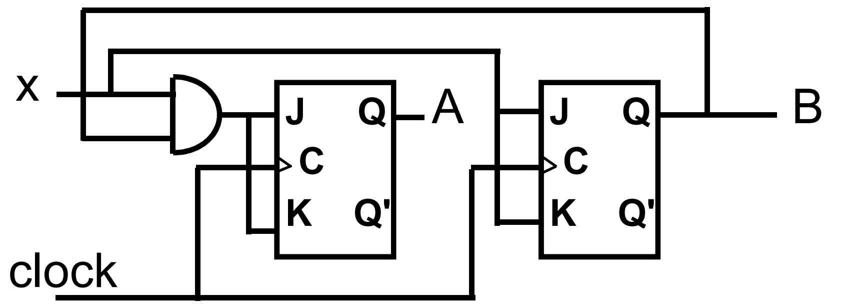
Example: 2-bit Counter -> 2 FF's



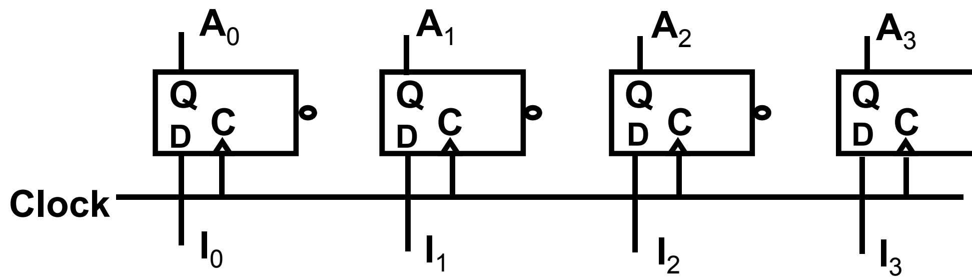
current state		input	next state		FF inputs			
A	**B**	**x**	**A**	**B**	**Ja**	**Ka**	**Jb**	**Kb**
0	0	0	0	0	0	d	0	d
0	0	1	0	1	0	d	1	d
0	1	0	0	1	0	d	d	0
0	1	1	1	0	1	d	d	1
1	0	0	1	0	d	0	0	d
1	0	1	1	1	d	0	1	d
1	1	0	1	1	d	0	d	0
1	1	1	0	0	d	1	d	1



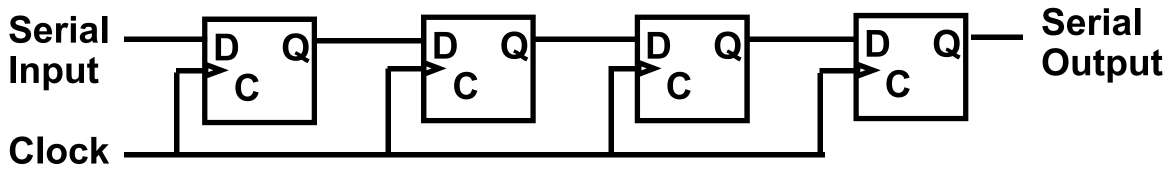
Ja = Bx Ka = Bx Jb = x Kb = x



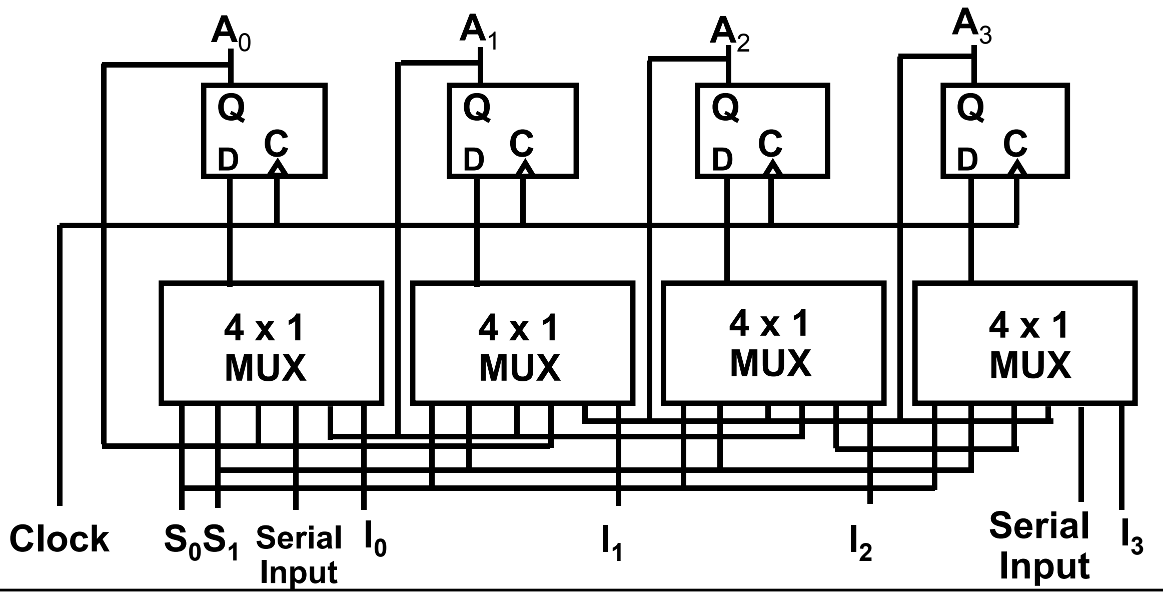
SEQUENTIAL CIRCUITS - Registers



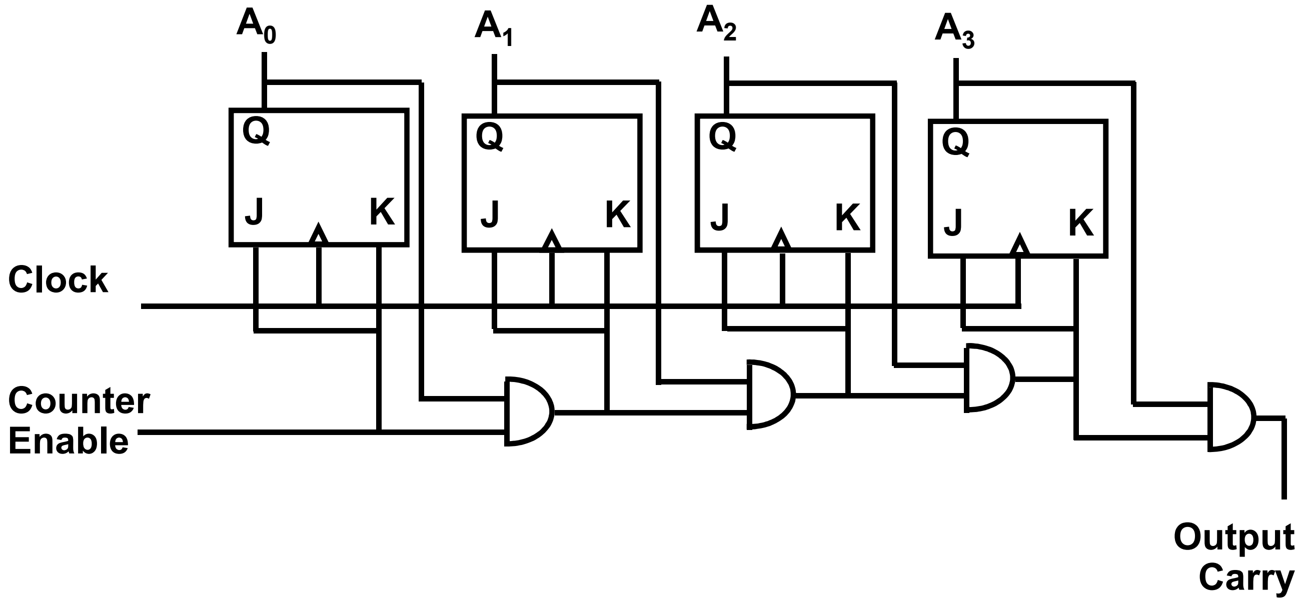
Shift Registers



Bidirectional Shift Register with Parallel Load

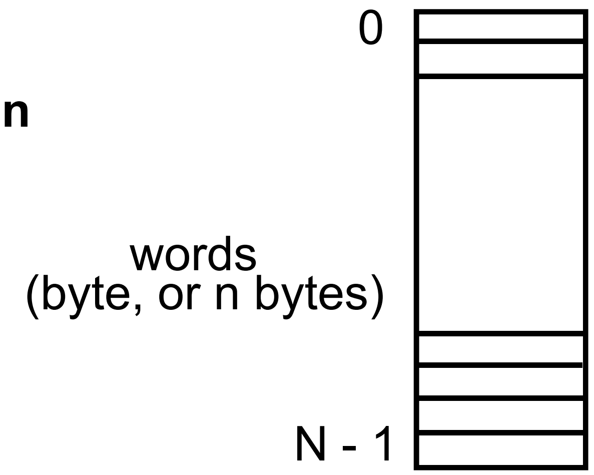


SEQUENTIUAL CIRCUITS - Counters



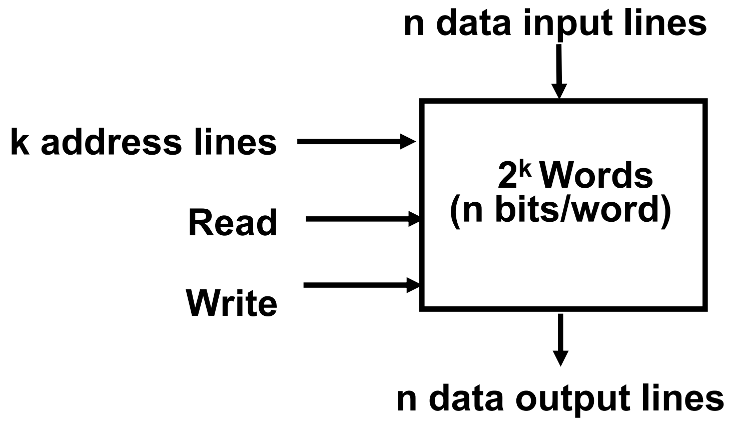
MEMORY COMPONENTS

Logical Organization



Random Access Memory

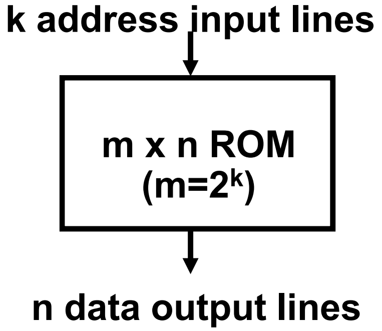
- **Each word has a unique address**
- **Access to a word requires the same time independent of the location of the word**
- **Organization**



READ ONLY MEMORY(ROM)

Characteristics

- Perform read operation only, write operation is not possible
- Information stored in a ROM is made permanent during production, and cannot be changed
- Organization



Information on the data output line depends only on the information on the address input lines.

--> Combinational Logic Circuit

$X_0=A'B' + B'C$
$X_1=A'B'C + A'BC'$
$X_2=BC + AB'C'$
$X_3=A'BC' + AB'$
$X_4=AB$

$$\Big\downarrow$$

Canonical minterms

$X_0=A'B'C' + A'B'C + AB'C$
$X_1=A'B'C + A'BC'$
$X_2=A'BC + AB'C' + ABC$
$X_3=A'BC' + AB'C' + AB'C$
$X_4=ABC' + ABC$

$$\longrightarrow$$

address ABC	*Output* X_0	X_1	X_2	X_3	X_4
000	1	0	0	0	0
001	1	1	0	0	0
010	0	1	0	1	0
011	0	0	1	0	0
100	0	0	1	1	0
101	1	0	0	1	0
110	0	0	0	0	1
111	0	0	1	0	1

TYPES OF ROM

ROM

- Store information (function) during production
- Mask is used in the production process
- Unalterable
- Low cost for large quantity production --> used in the final products

PROM (Programmable ROM)

- Store info electrically using PROM programmer at the user’s site
- Unalterable
- Higher cost than ROM -> used in the system development phase
  -> Can be used in small quantity system

EPROM (Erasable PROM)

- Store info electrically using PROM programmer at the user’s site
- Stored info is erasable (alterable) using UV light (electrically in some devices) and rewriteable
- Higher cost than PROM but reusable --> used in the system development phase. Not used in the system production due to eras ability

INTEGRATED CIRCUITS

Classification by the Circuit Density

SSI -	**several (less than 10) independent gates**
MSI -	**10 to 200 gates; Perform elementary digital functions;**
	Decoder, adder, register, parity checker, etc
LSI -	**200 to few thousand gates; Digital subsystem**
	Processor, memory, etc
VLSI -	**Thousands of gates; Digital system**
	Microprocessor, memory module

Classification by Technology

TTL -	**Transistor-Transistor Logic**
	Bipolar transistors
	NAND
ECL -	**Emitter-coupled Logic**
	Bipolar transistor
	NOR
MOS -	**Metal-Oxide Semiconductor**
	Unipolar transistor
	High density
CMOS -	**Complementary MOS**
	Low power consumption