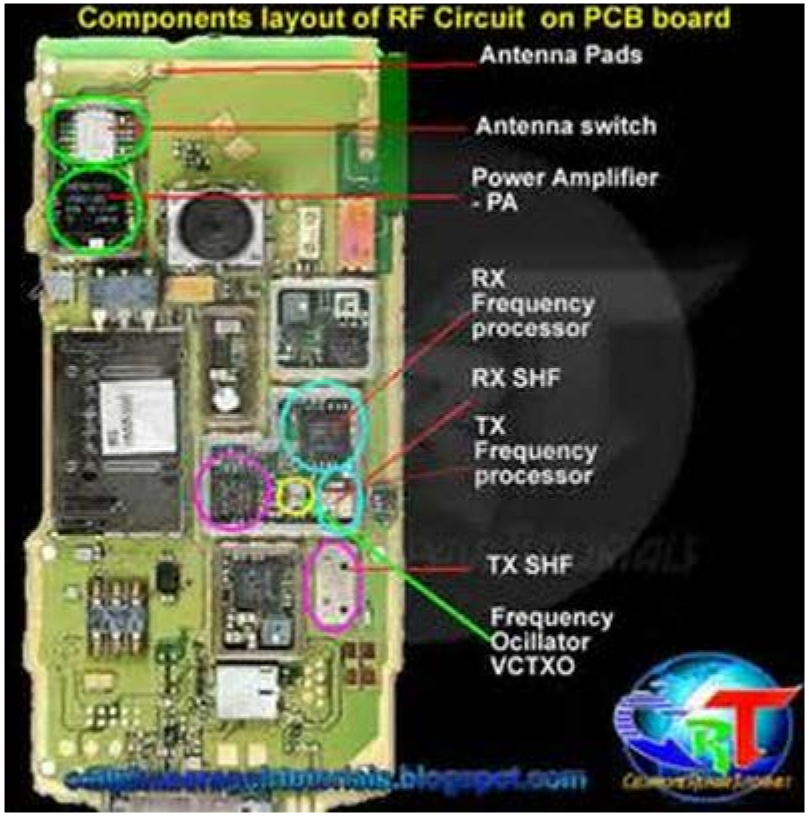
Introduction to Electronics

Digital Electronics

Introduction to Digital Electronics

Signal is represented by discrete bands.



Internal circuitry of a mobile phone.

An industrial digital controller.

Intel core i9.

Introduction to Digital Electronics

Signal is represented by discrete bands.

Advantages:

- Digital signal can be transmitted without degradation due to noise.
- More precise representation of a digital signal: use more binary digits, requires more digital circuits. An analog system requires fundamental improvements in the linearity and noise characteristics of each step of the signal chain.
- Computer-controlled digital systems can be controlled by software, allowing new functions to be added without changing hardware.
- Information storage can be easier in digital systems than in analog (provided total noise is below a certain level).
- Error correction.

Disadvantages:

- Usually, use more energy to accomplish the same tasks.
- High frequency limitation: cellular telephones still use a low-power analog front-end. Digital circuits are sometimes more expensive, especially in small quantities.
- Quantization error.
- In some schemes, a single bit error can damage a large amount of data.

Introduction to Binary Logic

- AND: x AND y →x.y (output is true only if all of the inputs are true).
- OR: x OR y →x+y (output is true if any of the inputs is true).
- NOT: x not →$\bar{y}$ or sometimes y’ (the output is opposite of that of the input).

AND

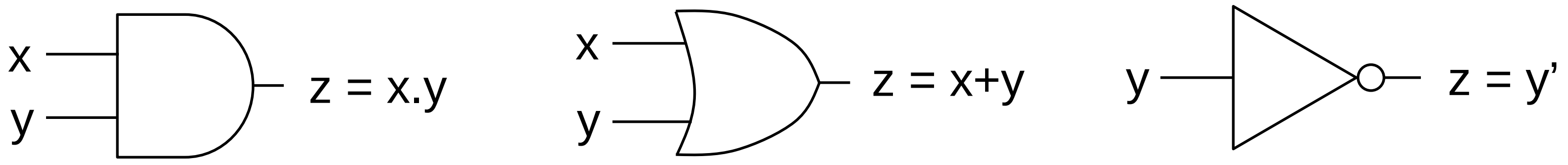
x	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x.y
0	0	0
0	1	1
1	0	1
1	1	1

NOT

y	y'
0	1
1	0



Symbols for digital logic circuits.

- Multiple input gates: z = A.B.C.D

Binary Logic

NAND

x	y	(x.y)’
0	0	1
0	1	1
1	0	1
1	1	0

NOR

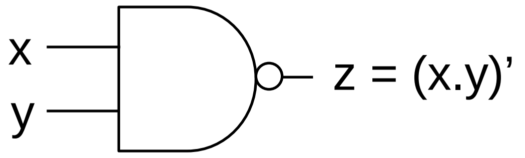
x	y	(x+y)’
0	0	1
0	1	0
1	0	0
1	1	0

X-OR

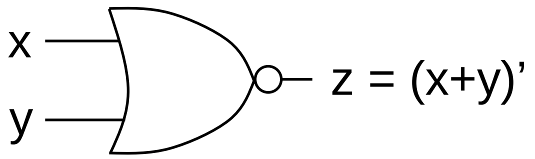
x	y	
0	0	0
0	1	1
1	0	1
1	1	0

X-NOR

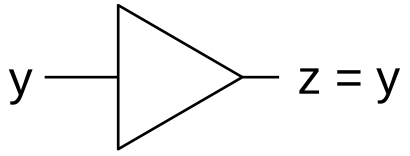
x	y	
0	0	1
0	1	0
1	0	0
1	1	1



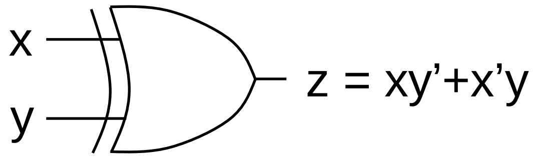
NAND.



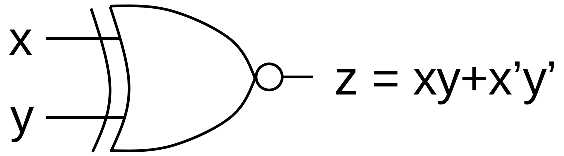
NOR.



Buffer.



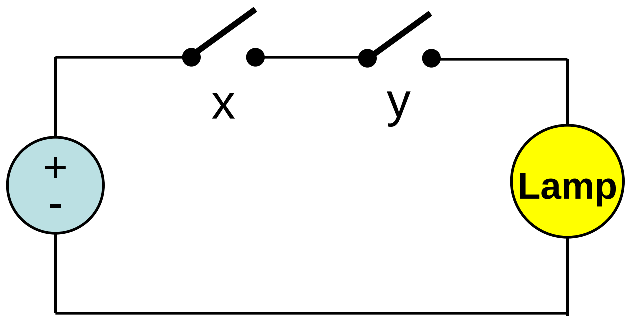
X-OR.



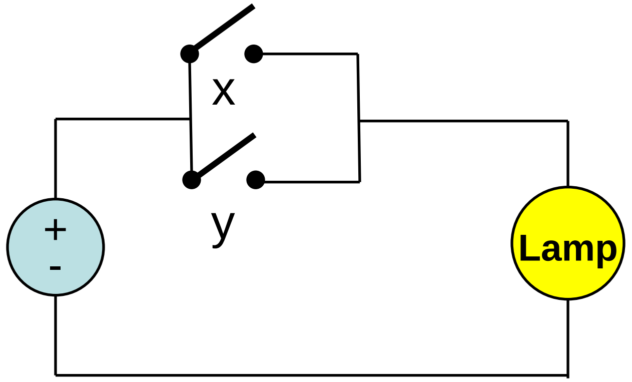
X-NOR.

Symbols for digital logic circuits.

Binary Logic Implementation



Realization of two input AND using switches.

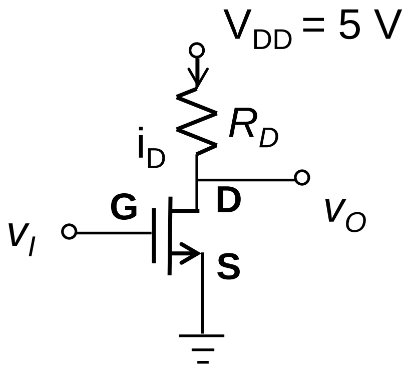


Realization of two input OR using switches.

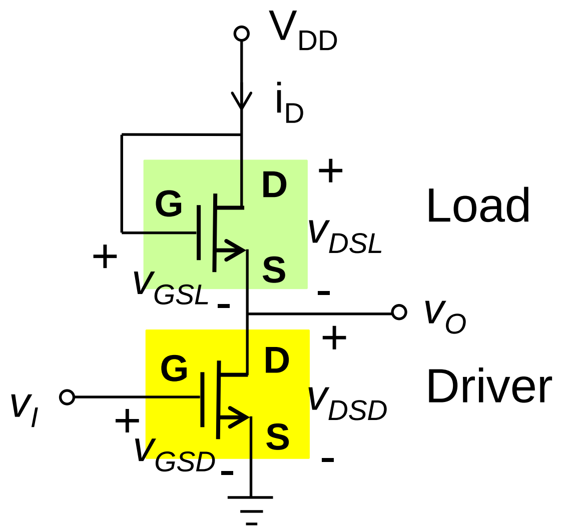
- Switches can be realized by diode and transistor – DTL logic gates.
  by transistors – TTL logic gates.
- Some important parameters: fan-out, power dissipation, propagation delay, noise margin.

IC family	Voltage supply	High level		Low level	
		Range	Typical	Range	Typical
TTL	V_{CC} = 5 V	2.4-5	3.5	0-0.4	0.2
CMOS	V_{DD} = 3-10 V	V_{DD}	V_{DD}	0-0.5	0

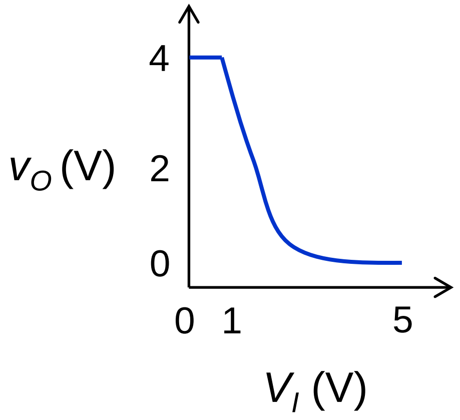
Binary Logic Implementation (NMOS)



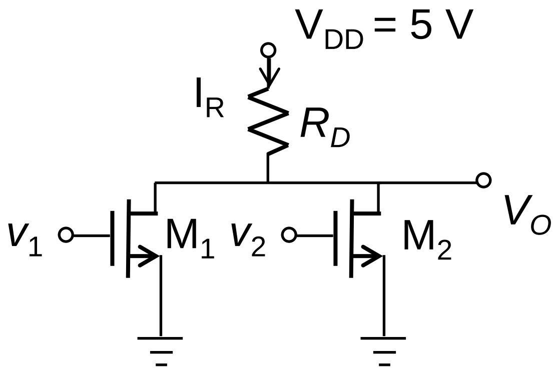
NMOS inverter.



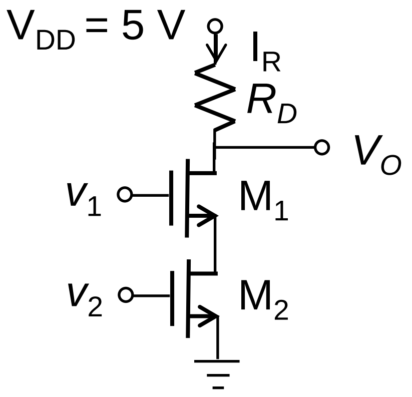
Inverter with an NMOS as a load.



Transfer characteristics.



Two input NOR gate.



Two input NAND gate.

Boolean Algebra

- Introduced by George Boole in 1854.
- Two elements: 0, 1.
- Two binary operators: +, . (no subtraction or division)
- A new operator: complement.

Basic theorems and postulates:

$x + 1 = 1$	$x.1 = x$
$x + 0 = x$	$x.0 = 0$
$x + x = x$	$x.x = x$
$x + x' = 1$	$x.x' = 0$
$(x')' = x$	$x.y = y.x$ (commutative)
$x + y = y + x$	$x.(x + y) = x + x.y$ (distributive)

DeMorgan's theorem:

1. $(x + y)' = x'.y'$ $\Rightarrow$ NOR
2. $(x.y)' = x' + y'$ $\Rightarrow$ NAND

Boolean Algebra

Using truth tables, prove that

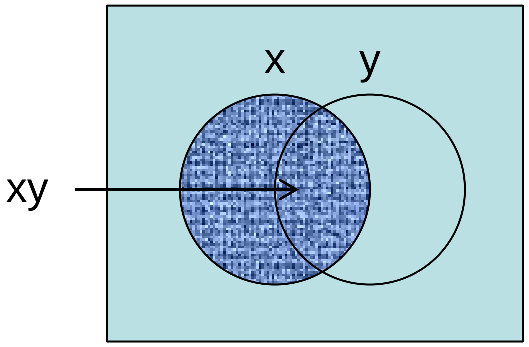
1. $(x + y)' = x'y'$.
2. $x'y + x = x + y$.

Solutions:

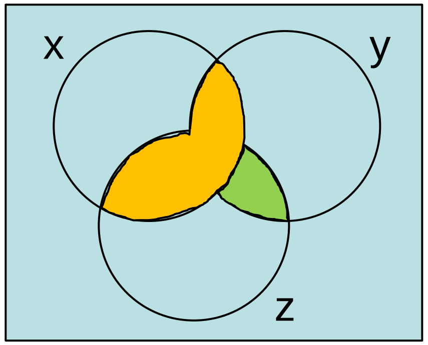
x	y	(x+y)'	x'y'
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Venn diagram:

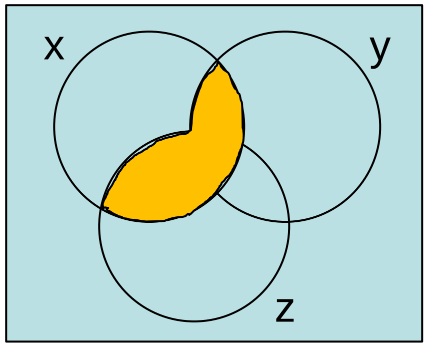
- Circles are drawn inside a rectangle. One for each variable.
- The value of the variable is 1 inside the corresponding circle, and 0 outside.
- AND: intersection, OR: union.



xy + x = x



x(y + z)



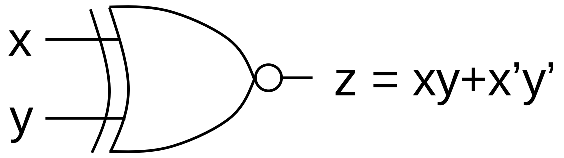
xy + xz

Boolean Algebra

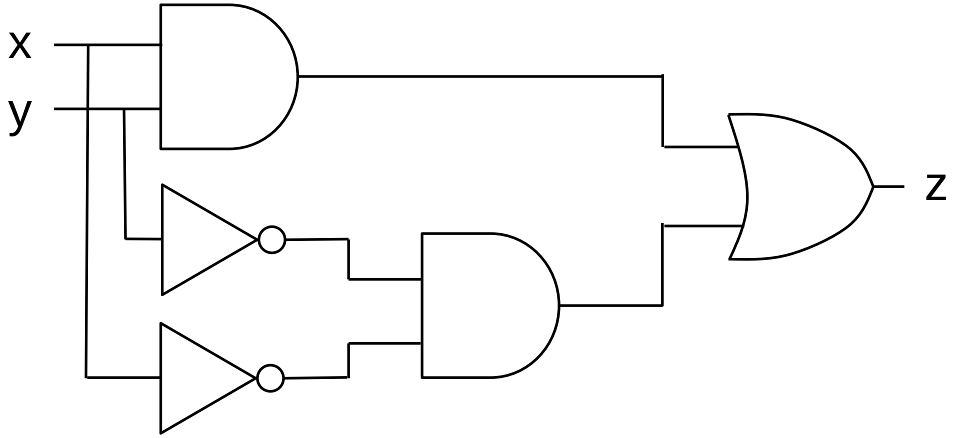
Realization of a function using logic gates:

1. $(x + y')(x\,y + x')$

$= x\,y + x'y'$



Implementation using X-NOR gate.



Implementation using AND, OR and NOT gates.

Simplify the following functions: $x\,y + x'z + yz$

Solutions:

$$
\begin{aligned}
& x\,y + x'z + yz \\
&= x\,y + x'z + yz(x + x') \\
&= x\,y + x'z + xyz + x'yz \\
&= xy(1 + z) + x'z(1 + y) \\
&= xy + x'z
\end{aligned}
$$

Minterms and Maxterms

Minterms

x	y	z	Term	Designation
0	0	0	x‘y’z’	m_0
0	0	1	x’y’z	m_1
0	1	0	x’yz’	m_2
0	1	1	x’yz	m_3
1	0	0	xy’z’	m_4
1	0	1	xy’z	m_5
1	1	0	xyz’	m_6
1	1	1	xyz	m_7

Take the minterms for which the function is 1.

Maxterms

x	y	z	Term	Designation
0	0	0	x+y+z	M_0
0	0	1	x+y+z’	M_1
0	1	0	x+y’+z	M_2
0	1	1	x+y’+z’	M_3
1	0	0	x'+y+z	M_4
1	0	1	x'+y+z’	M_5
1	1	0	x'+y’+z	M_6
1	1	1	x'+y’+z’	M_7

Take the maxterms for which the function is 0.

- An *n* variable binary number will have 2^n minterms/maxterms.
- Any Boolean function can be expressed by the OR operation of all the minterms.
- Any Boolean function can be expressed by the AND operation of all the maxterms.

Minterms and Maxterms

Truth table for two functions f_1 and f_2

x	y	z	f_1	f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Representation in terms of minterms:

$$f_1 = x'y'z + x'yz' + x'yz \qquad = \sum(1, 2, 3)$$

$$f_2 = x'yz + xy'z + xyz' + xyz \qquad = \sum(3, 5, 6, 7)$$

Representation in terms of maxterms:

$$f_1 = (x+y+z)(x'+y+z)(x'+y+z')(x'+y'+z)(x'+y'+z') \qquad = \prod(0, 4, 5, 6, 7)$$

$$f_2 = (x+y+z)(x+y+z')(x+y'+z)(x'+y+z) \qquad = \prod(0, 1, 2, 4)$$

Minterms and Maxterms

- Express the Boolean function F = A+B'C in a sum of minterms form.

Solutions:

- Consider each term individually.
- If *x* is the missing variable, multiply the term by (*x*+*x*')

$$A = A(B + B') = AB + AB'$$

$$A = (AB + AB')(C + C')$$

$$= ABC + ABC' + AB'C + AB'C'$$

$$B'C = B'C(A + A')$$

$$= AB'C + A'B'C$$

$$\therefore F = A'B'C + AB'C' + AB'C + ABC' + ABC$$

- Express the Boolean function F = xy + x'y' + yz in a sum of minterms form.

Solutions:

$$F = xyz + xyz' + x'yz + x'y'z + x'y'z'$$

Minterms and Maxterms

- Express the Boolean function F = AB+A'C in a product of maxterms form.

Solutions:

- Use the distributive law: x + yz = (x+y)(x+z).
- For any missing variable x, ORed with xx' and again use the distributive law.

$$\begin{aligned} AB + A'C &= (AB + A')(AB + C) \\ &= (A + A')(B + A')(A + C)(B + C) \\ &= (A' + B)(A + C)(B + C) \end{aligned}$$

$$(A' + B) = (A' + B) + CC' = (A' + B + C)(A' + B + C')$$

$$(A + C) = (A + C) + BB' = (A + B + C)(A + B' + C)$$

$$(B + C) = (B + C) + AA' = (A + B + C)(A' + B + C)$$

$$\therefore F = (A + B + C)(A + B' + C)(A' + B + C)(A' + B + C')$$

- Express the Boolean function F = xy + x'y' + yz in a product of maxterms form.

Solutions:

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

Simplification of Boolean Functions

- A straight forward method is by using Karnaugh map or Veitch diagram.

Two variable K-map using the minterms

x\y	0	1
0	m_0	m_1
1	m_2	m_3

x\y	0	1
0	*x'y'*	*x'y*
1	*xy'*	*xy*

Representation of a function F = x'y+xy'

x\y	0	1
0	0	1
1	1	0

Three variable K-map using the minterms

x\yz	00	01	11	10
0	*x'y'z'*	*x'y'z*	*x'yz*	*x'yz'*
1	*xy'z'*	*xy'z*	*xyz*	*xyz'*

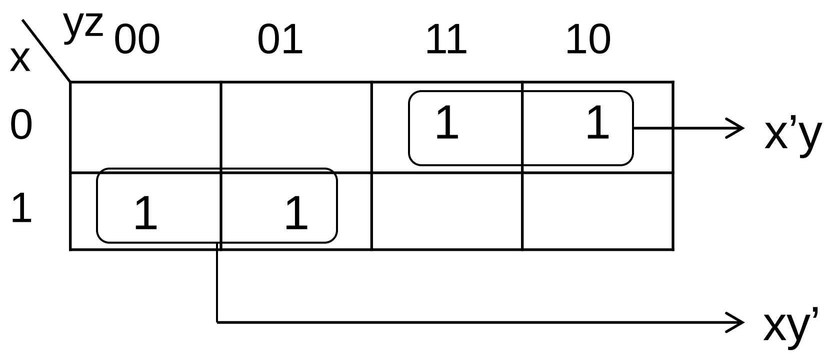
Representation of a function F = x'yz+xy'z+xyz

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

- A *n* variable K-map consist of 2^n squares.
- The minterms are not in binary sequence, only one bit changes in the listing sequence.

Simplification of Boolean Functions

Simplify the function F = x’yz+x’yz’+xy’z’+xy’z.



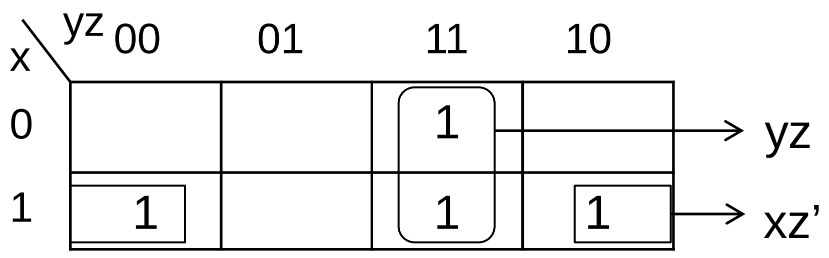
F = x’y+xy’

Sum of product simplification steps:

- The squares representing the minterms mark as 1.
- Subdivide the area: put 2^n adjacent cells containing 1 into groups, try to maximize the number of cells in a group. A single 1 can be a common element to multiple groups.
- While grouping into squares, consider that the opposite sides and opposite corners of a K-map are joined together i.e. as if any K-map is drawn on a spherical surface.
- A square containing 1 provides only one term: discard the variable/variables which changes bit within a group, the remaining variables will provide the term.

Simplification of Boolean Functions

Simplify the function F = x’yz+xy’z’+xyz’+xyz.



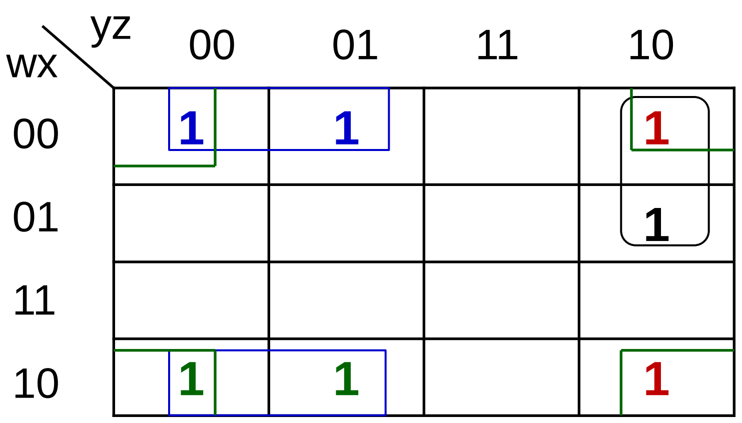
x \ yz	00	01	11	10
0			1	
1	1		1	1

F = xz’+yz

Four variable K-map using the minterms

wx \ yz	00	01	11	10
00	*w’x’y’z’*	*w’x’y’z*	*w'x’yz*	*w'x’yz’*
01	*w'xy‘z’*	*w'xy’z*	*w'xyz*	*w'xyz’*
11	*wxy’z’*	*wxy’z*	*wxyz*	*wxyz'*
10	*wx‘y’z’*	*wx‘y’z*	*wx‘yz*	*wx‘yz’*

Representation of a function
F = w’x’y’+x’yz’+w’xyz’+wx’y’



wx \ yz	00	01	11	10
00	1	1		1
01				1
11				
10	1	1		1

F = x’y’ + x’z’ + w’yz’.

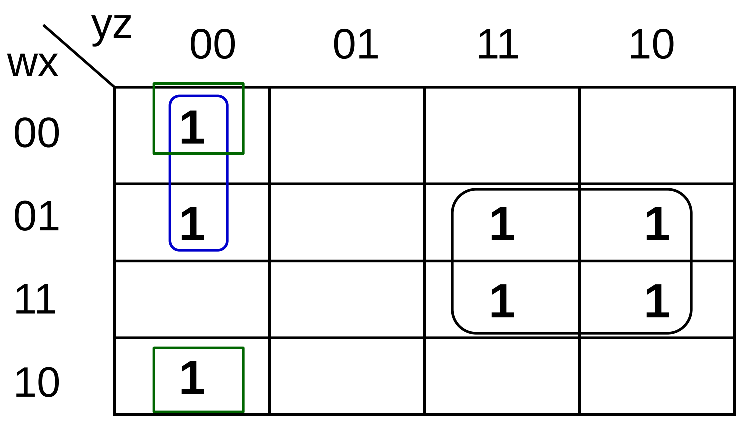
Simplification of Boolean Functions

Simplify the following Boolean functions:

1. $F1 = \sum(0, 4, 6, 7, 8, 14, 15)$

2. $F2 = w'x'y'z' + w'x'yz' + wx'yz' + wx'y'z' + w'xy'z + w'xyz + wxy'z + wxyz$

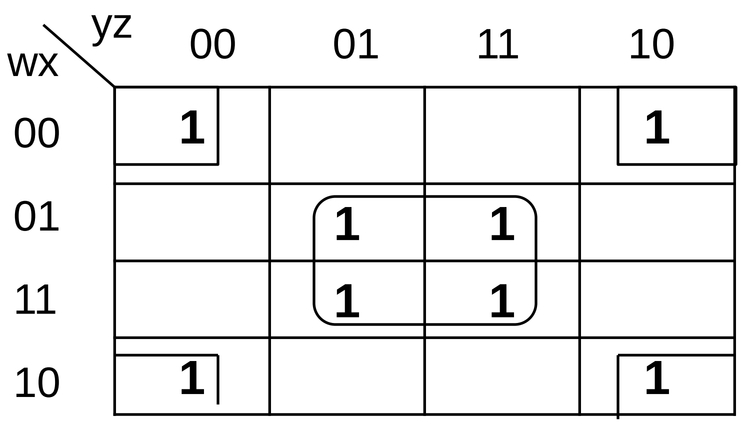
1. Representation of the function F1



wx \ yz	00	01	11	10
00	1			
01	1		1	1
11			1	1
10	1			

F = xy + w'y'z' + x'y'z'.

2. Representation of the function F2.



wx \ yz	00	01	11	10
00	1			1
01		1	1	
11		1	1	
10	1			1

F = xz + x'z'.

Simplification of Boolean Functions

Product of sum simplification:

Simplify the following Boolean functions:

1. $F1 = \sum(0, 4, 6, 7, 8, 14, 15)$
2. $F2 = \sum(0, 2, 5, 7, 8, 10, 13, 15)$

1. Representation of the function F1

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

F = (y+z')(w'+x'+y)(x+y').

2. Representation of the function F2.

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

F = (x'+z)(x+z').

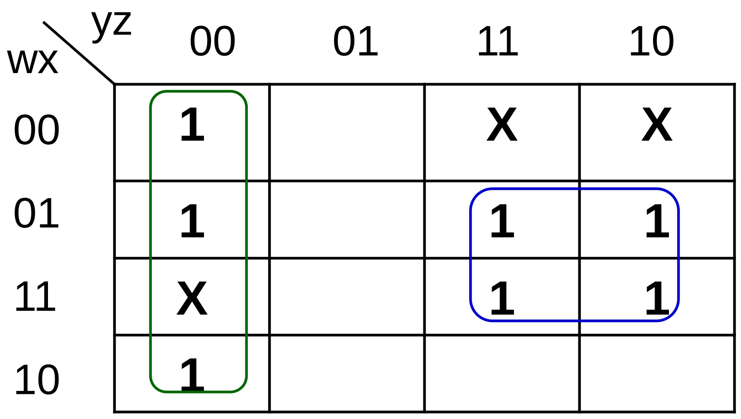
Simplification of Boolean Functions

Don’t care condition:

- In some applications, certain combinations of input variables **never occur**.
- The function value cannot be taken as 1 or 0 for those input combinations, they are represented by X and called the don’t care condition.
- When choosing adjacent squares for simplification, the X’s may be assumed as 0 or 1 or just ignore them, whichever gives the simplest expression.

Simplify the function $F1 = \sum(0, 4, 6, 7, 8, 14, 15)$ with $d = \sum(2, 3, 12)$

1. Representation of a function F1



wx \ yz	00	01	11	10
00	**1**		**X**	**X**
01	**1**		**1**	**1**
11	**X**		**1**	**1**
10	**1**			

F = y’z’ + xy

Simplify the following function:

$F2 = \sum(1, 3, 7, 11, 15)$ with $d = \sum(0, 2, 5)$

Solution: $F2$ **= w’z + yz**

Simplification of Boolean Functions

Don’t care condition:

Simplify the functions and express them in product of sum form

1. $F1 = \sum(0, 4, 6, 7, 8, 14, 15)$ with $d = \sum(2, 3, 12)$
2. $F2 = \sum(1, 3, 7, 11, 15)$ with $d = \sum(0, 2, 5)$

1. Representation of a function F1

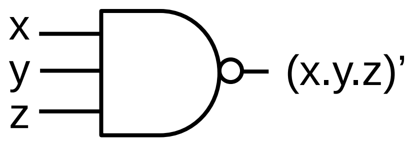
wx \ yz	00	01	11	10
00	**1**	**0**	**X**	**X**
01	**1**	**0**	**1**	**1**
11	**X**	**0**	**1**	**1**
10	**1**	**0**	**0**	**0**

F = (y+z’)(x+y’)

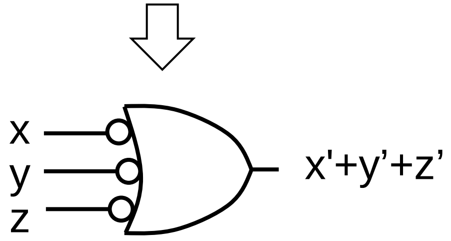
Solution: $F2 = z(w' + y)$

NAND and NOR Implementation

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

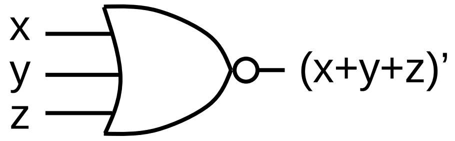


NAND (AND-invert).

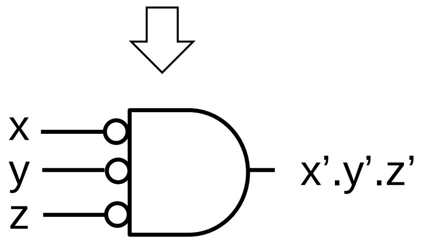


Invert-OR.

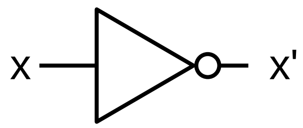
$F = (x + y + z)' = x'y'z'$



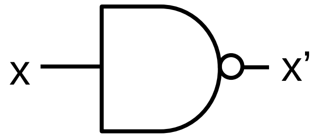
NOR (OR-invert).



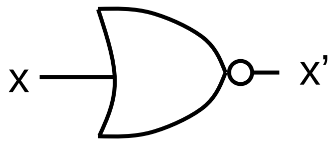
Invert-AND.



NOT (Buffer-invert).



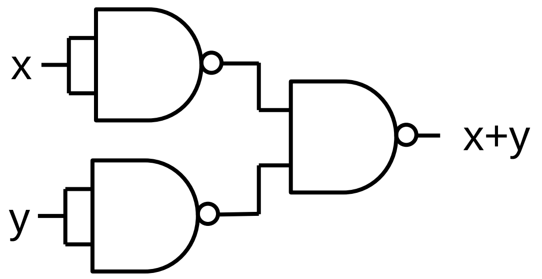
AND-invert.



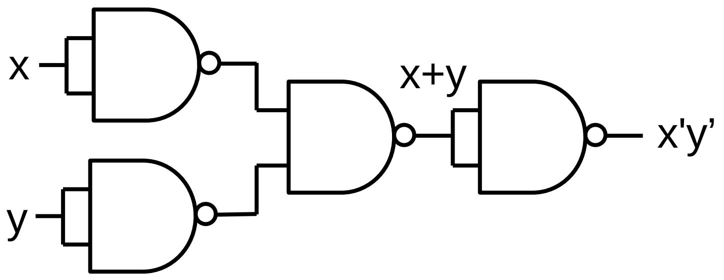
OR-invert.

NAND Gate as Universal Gate

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

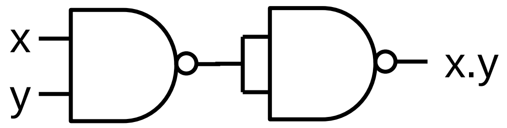


OR gate.

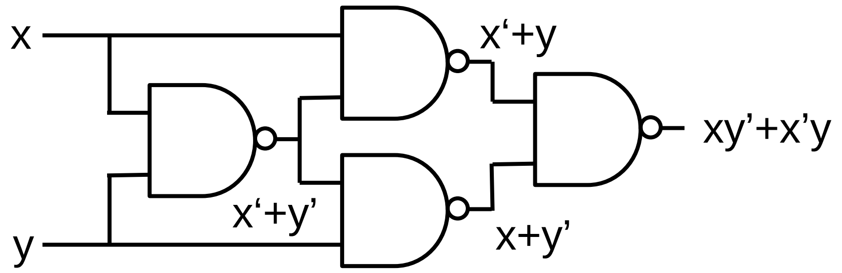


NOR gate.

$$F = xy = ((xy)')'$$



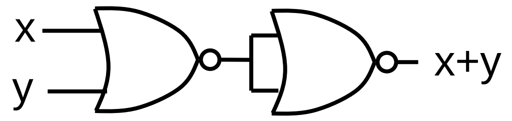
AND gate.



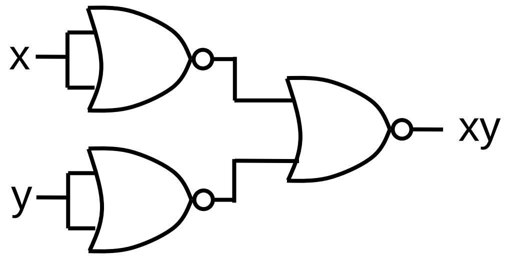
X-OR gate.

X-NOR: use one more NAND gate.

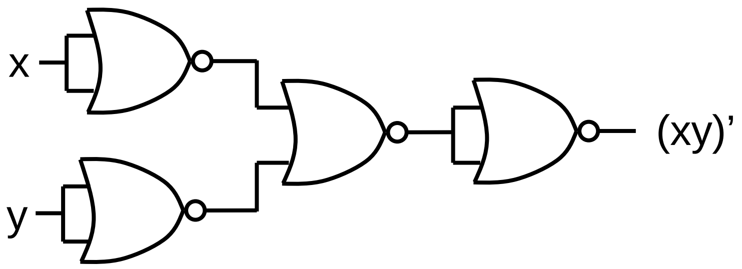
NOR Gate as Universal Gate



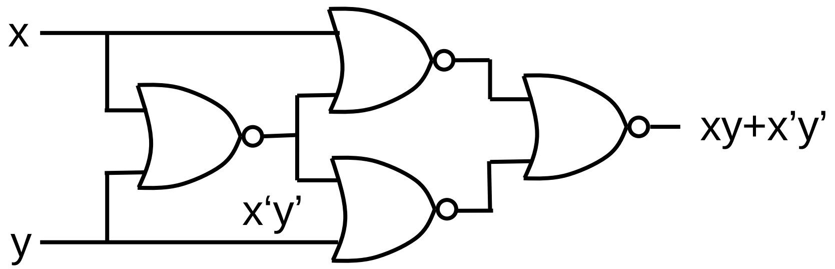
OR gate.



AND gate.



NAND gate.



X-NOR gate.

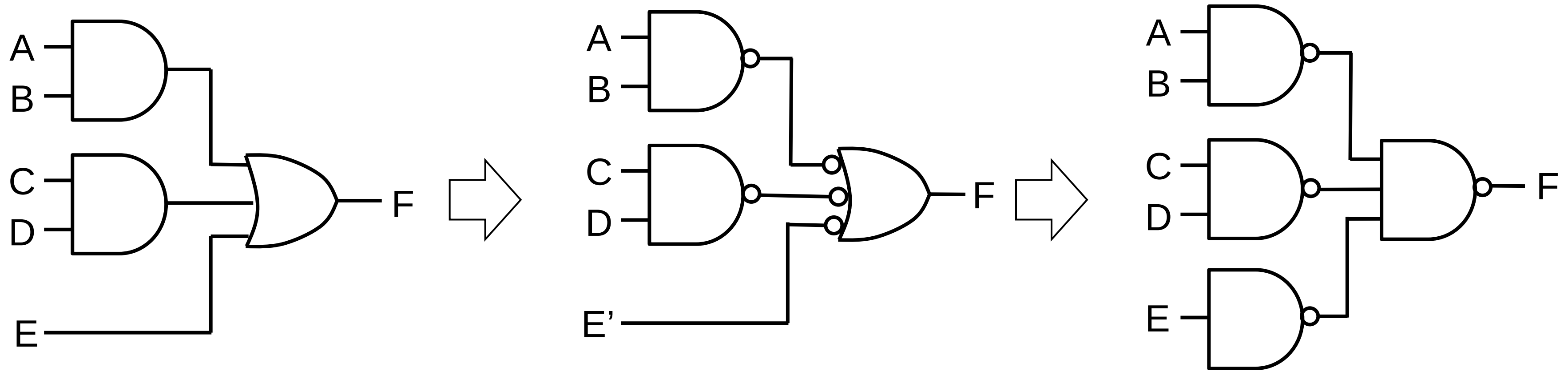
X-OR: use one more NAND gate.

NAND Implementation

1. Steps:

- Represent the simplified function in sum of product form.
- Use two inverters in between the AND and OR gate.
- Convert invert-OR to NAND.

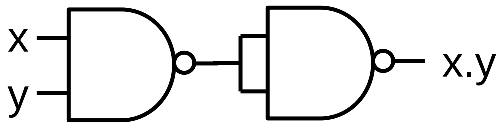
Simplify the function F = AB + CD + E



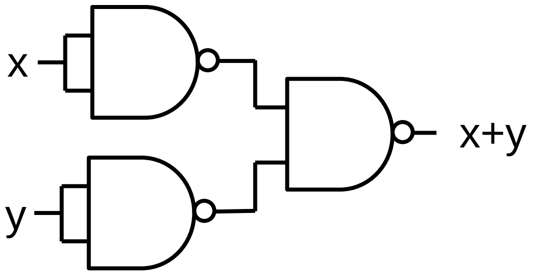
NAND Implementation

2. Steps:

- Represent the simplified function in sum of product form and implement using the AND, OR and NOT gates.
- Replace all AND, OR and NOT gates by their equivalent NAND representation.
- Remove cascaded inverter if any.

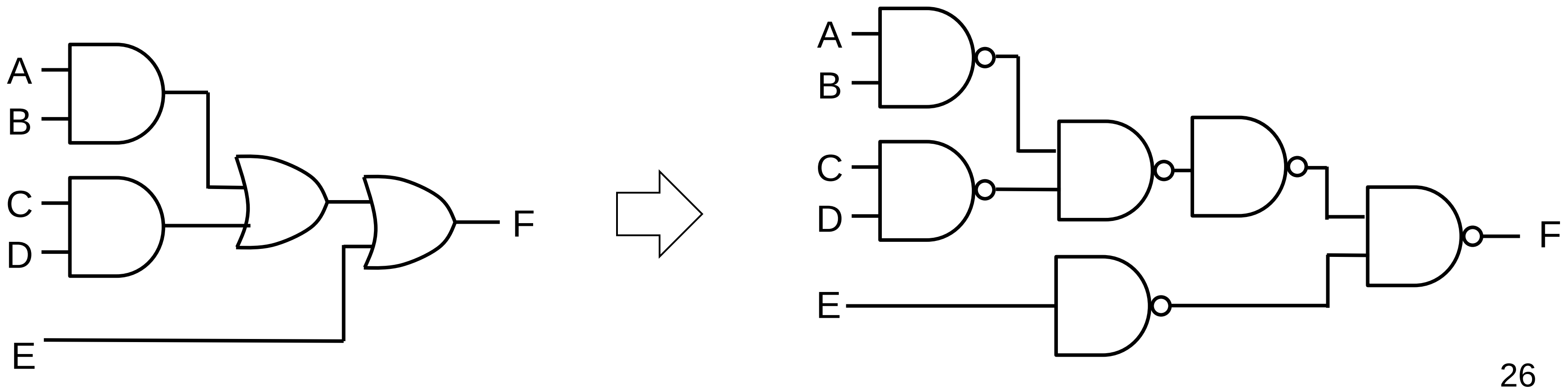


AND



OR

Implement the function F = AB + CD + E



Combinational Logic

Logic circuits:

1. Combinational circuit: at any time, the outputs are determined directly from the present inputs.
2. Sequential circuit: the outputs depend not only on present inputs, but also on past inputs (memory). Circuit behavior must be specified by a time sequence of inputs and internal states.

Implementation using combinational circuits:

1. Determination of the number of inputs and outputs is required.
2. Obtain the truth table/ their relationship and from that represent them in a function form.
3. Simplify the function.
4. Implement by using AND, OR and NOT gates.
5. Depending on availability, convert to NAND/ NOR logic.
6. For multiple functions, if possible, try to express an output in terms of other outputs.

Combinational Logic Adders

Adders:

The basic arithmetic operation.

Half-adder: perform addition of two bits.

Full-adder: perform addition of two bits and also take care of a previous carry if any.

Half-adder:

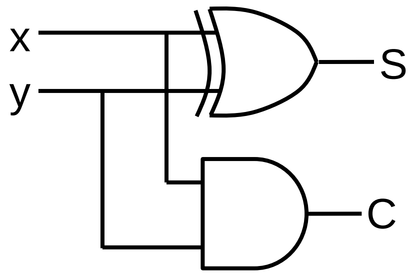
Truth table

x	*y*	*C*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

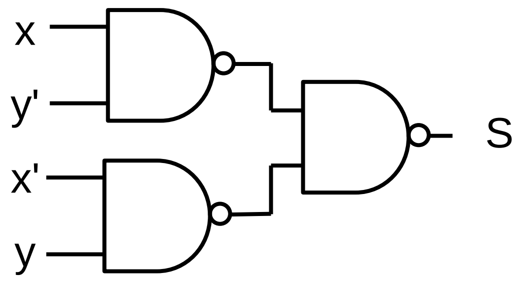
The sum (S) and carry (C) bits can be express as follows:

$$S = x'y + xy' = x \oplus y = (x + y)(x' + y')$$

$$C = xy$$



Half-adder implementation.



NAND gate implementation of the sum bit.

Combinational Logic Adders

Full-adder:

x and y are the inputs and z denotes the LSB of previous carry.

Carry	1
Augend	57
Addend	47
Sum	104

Truth table

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

Solve for sum bit.

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

$$S = x'y'z + x'yz' + xy'z' + xyz.$$

Solve for carry bit.

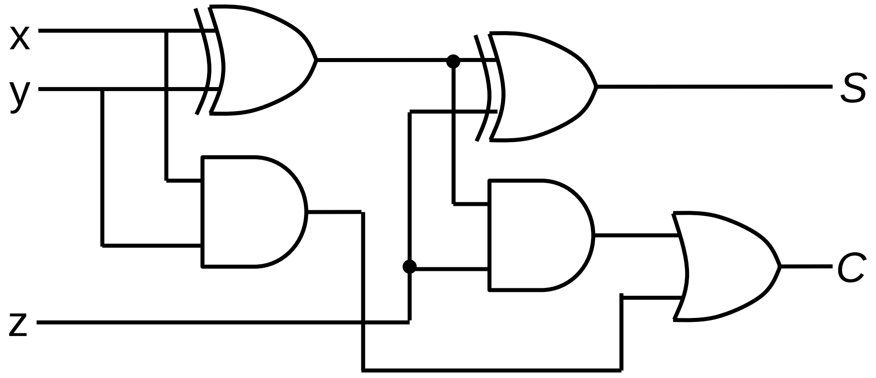
x \ yz	00	01	11	10
0			1	
1		1	1	1

$$C = xy + yz + zx$$

Combinational Logic Adders

Full-adder implementation:

$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= z'(xy' + x'y) + z(xy + x'y')$$
$$= xy'z' + x'yz' + xyz + x'y'z$$
$$C = z(x'y + xy') + xy = xy'z + x'yz + xy$$



Full-adder implementation.

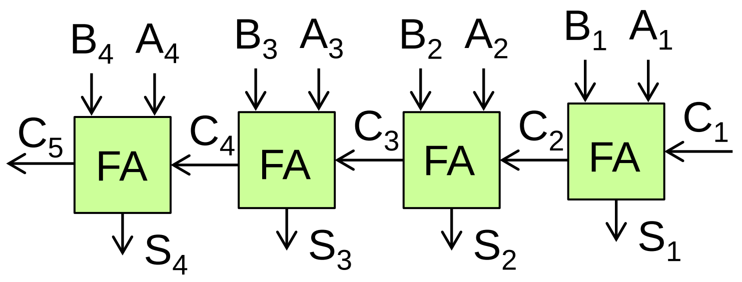
Binary parallel adder:

- Add two binary numbers of *n* bits.
- An *n* bit parallel adder requires *n* number of full adders.
- Starting from the LSB, connect the inputs and the carry sequentially.
- Serial adder can be designed by using a single full adder and memory elements.

Combinational Logic

Example:

Subscript *i*	4321	
Input carry	0110	C_i
Augend	1011	A_i
Addend	0011	B_i
Sum	1110	S_i
Output carry	0011	C_{i+1}



A four bit parallel adder.

Carry propagation:

- Each gate takes finite time to respond: propagation delay Δt.
- The (n+1)th carry is available only after $n\times \Delta t$ (one solution is look-ahead carry).

Subtractors:

Subtraction can be accomplished by taking the complement of the subtrahend and adding it to the minuend.

Here, it will be implement directly.

Half-subtractor: perform subtraction of two bits.

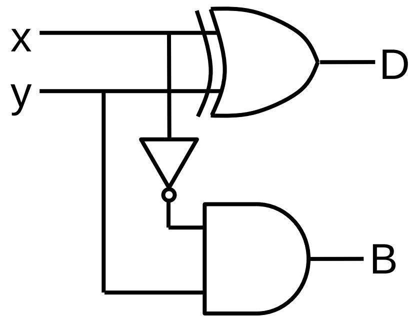
Full-subtractor: perform subtraction of two bits and also take care of a previous borrow if any.

Combinational Logic

Half-subtractor:

- x<y : borrow a 1 from the next higher state.

$$D = x'y + xy' \quad = S$$
$$B = x'y$$



Half-adder implementation.

Truth table

x	*y*	*B*	*D*
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Combinational Logic

Full-subtractor:

Truth table

x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

x: minuend, y: subtrahend, and z: previous borrow.
(x – y = D)

Solve for difference bit.

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

$$D = x'y'z + x'yz' + xy'z' + xyz.$$

Solve for borrow bit.

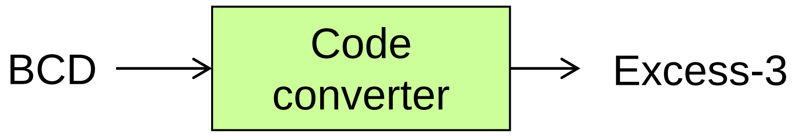
x \ yz	00	01	11	10
0		1	1	1
1			1	

$$C = x'y + yz + x'z$$

Q. Implement the full-subtractor by two input NAND gates only.

Combinational Logic

Code converter:



- Remaining six binary input combination never occur: don't care condition.

AB \ CD	00	01	11	10
00	1			1
01	1			1
11	X	X	X	X
10	1		X	X

z = D'.

$y = CD + C'D'$,

$x = B'C + B'D + BC'D'$,

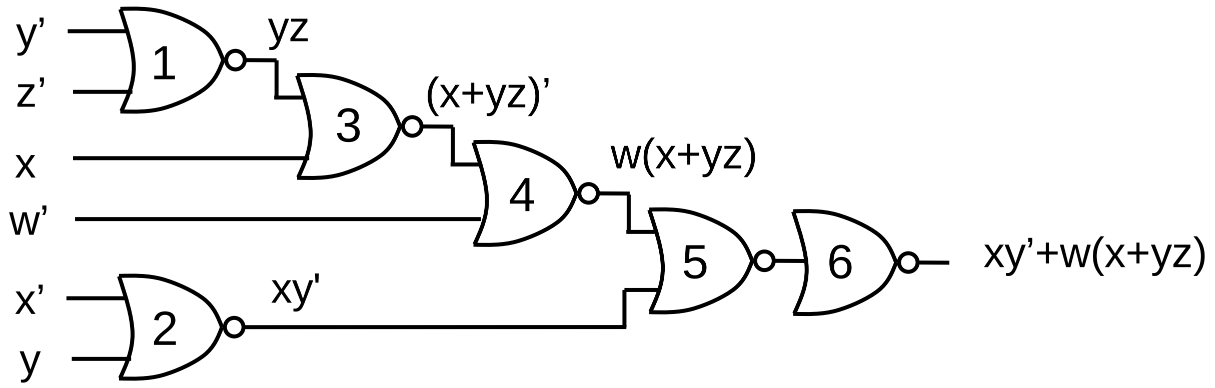
$w = A + BC + BD$.

Input BCD				Output Excess-3 code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Combinational Logic

Analysis of a given network:

1. Label the inputs and determine the number of input variables.
2. Obtain the Boolean functions for each gate and continue calculation until the outputs are obtained.
3. By repeated substitution of previously defined functions, obtain the output in terms of input variables only.



Analysis of the circuit.

Combinational Logic

X-OR and equivalence functions:

$$x \oplus y = x'y + xy',$$

$$x \odot y = xy + x'y' = \left(x \oplus y\right)',$$

$$\left(x \oplus y\right) \oplus z = x \oplus \left(y \oplus z\right) = x \oplus y \oplus z,$$

$$\left(x \oplus y \oplus z\right)' = x \oplus y \odot z,$$

$$\left(x \odot y \odot z\right) = x \odot y \oplus z.$$

wx \ yz	00	01	11	10
00		**1**		**1**
01	**1**		**1**	
11		**1**		**1**
10	**1**		**1**	

$w \oplus x \oplus y \oplus z$

- Very useful in error-detection and error-correction codes.
- *n* variable X-OR operation: $2^n/2$ minterms have an odd number of 1’s.
- *n* variable X-NOR operation: $2^n/2$ minterms have an even number of 1’s.

wx \ yz	00	01	11	10
00	**1**		**1**	
01		**1**		**1**
11	**1**		**1**	
10		**1**		**1**

$w \odot x \odot y \odot z$

Combinational Logic

Parity Checker:

- Parity bit: extra bit included with a binary message to make the number of 1's either odd (odd parity) or even (even parity).
- Parity generator: generates the parity bit in a transmitter.
- Odd parity generation: *P* = 1 if total number of 1's including *P* is odd.
- Parity checker: check the parity in a receiver.

$$P = x \oplus y \odot z$$
$$= x \odot y \oplus z.$$

$$C = x \odot y \odot z \odot P.$$

Odd-parity generation

Message			Parity bit
x	y	z	P
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Odd-parity check in a receiver.

Received bits				Error check
x	y	z	P	C
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

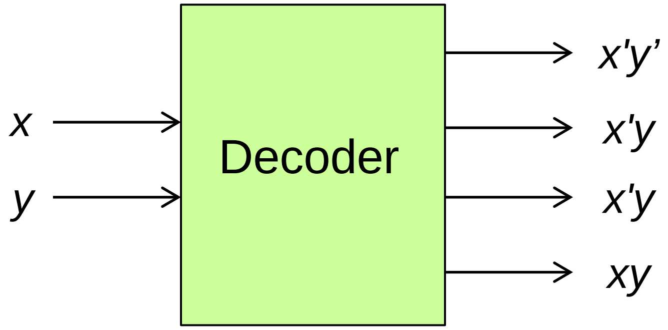
Combinational Logic

Decoders:

Converts binary information from n input lines to a maximum 2^n unique output lines.

A n-to-m line decoder ($m \le 2^n$) generates m unique output lines from n input lines.

If $m < 2^n$, rest ($2^n - m$) can be used as don’t care condition.



A 2-to-4 line decoder.

- Design a BCD-to-decimal decoder.

wx \ yz	00	01	11	10
00	D_0	D_1	D_3	D_2
01	D_4	D_5	D_7	D_6
11	X	X	X	X
10	D_8	D_9	X	X

- 10 AND gates and 4 NOT gates are required for implementation

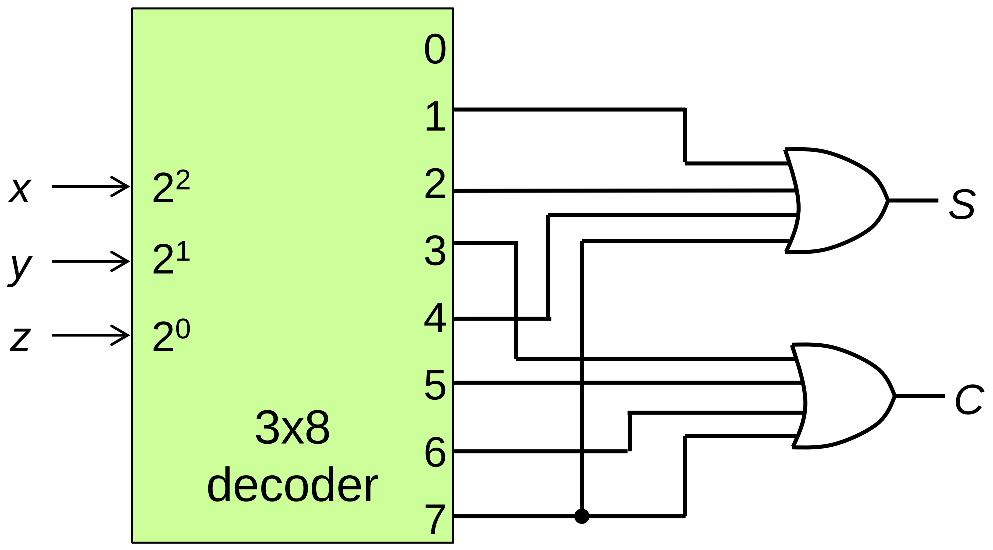
$$D_0 = w'x'y'z', \quad D_3 = x'yz, \quad D_6 = xyz', \quad D_9 = wz.$$
$$D_1 = w'x'y'z, \quad D_4 = xy'z', \quad D_7 = xyz,$$
$$D_2 = x'yz', \quad D_5 = xy'z, \quad D_8 = wz',$$

Combinational Logic

- Implement a full-adder circuit with a decoder and OR gates.

Solutions:

$$S(x,y,z)=\sum(1,2,4,7)$$
$$C(x,y,z)=\sum(3,5,6,7).$$

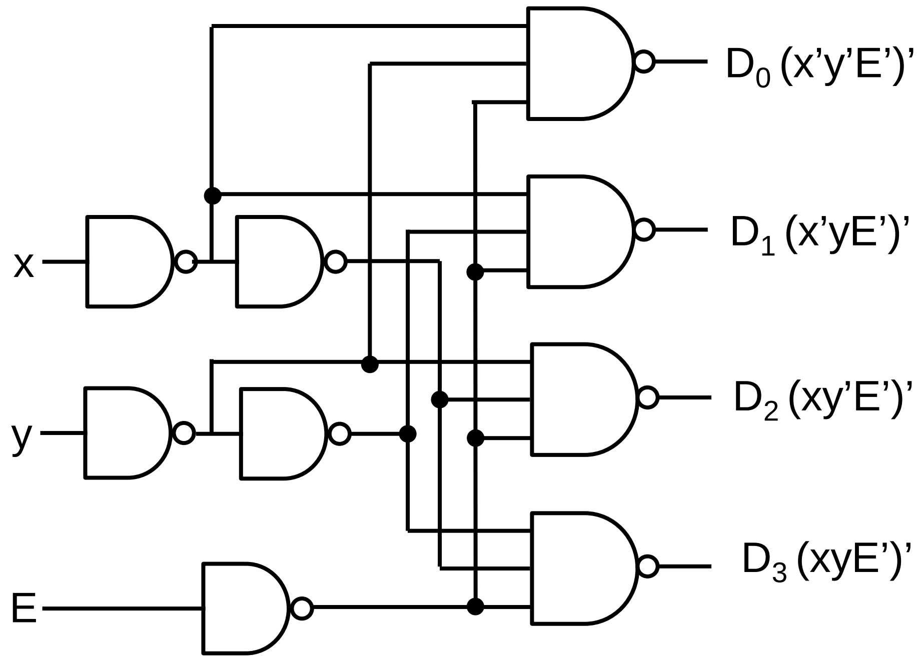


Full-adder with a decoder.

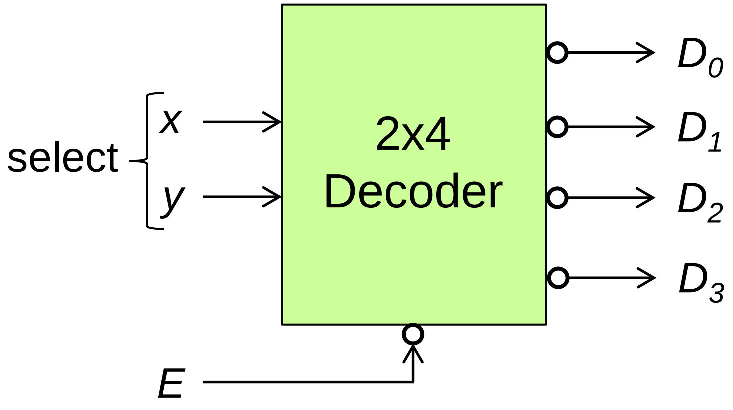
Combinational Logic

Demultiplexers:

- A demultiplexer receives information on a single line and transmits on one of 2^n possible output lines.
- A decoder with an enable input.
- Enable input: take AND operation with all outputs.



Implementation of a demultiplexer with NAND gates.

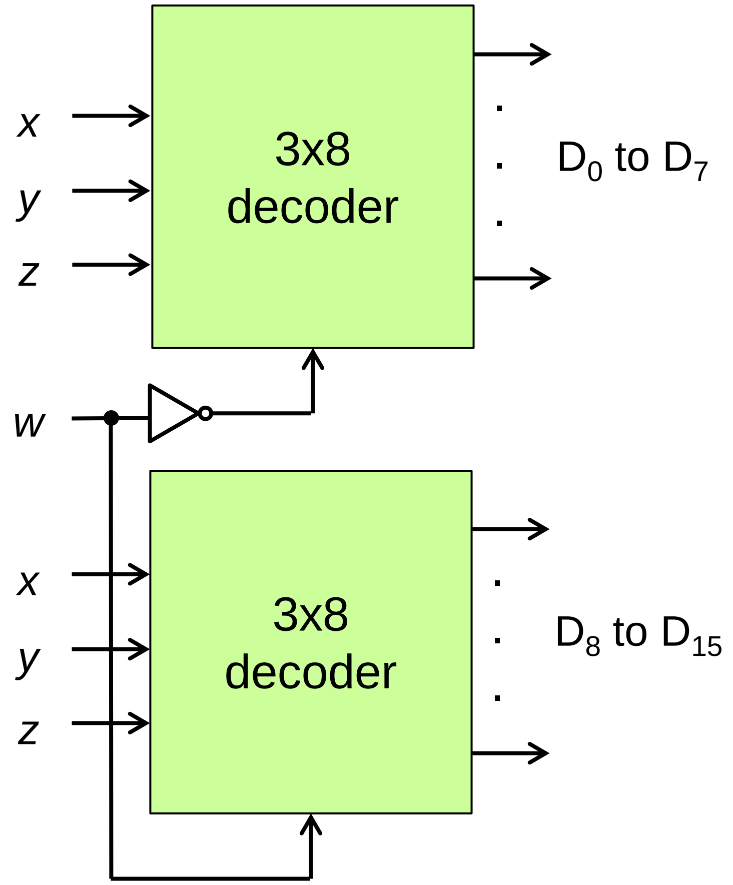


A 2-to-4 decoder with enable input.

Combinational Logic

Design a 4x16 demultiplexer using 3x8 demultiplexers.

Solution:



4x16 demultiplexer constructed with two 3x8 demultiplexer.

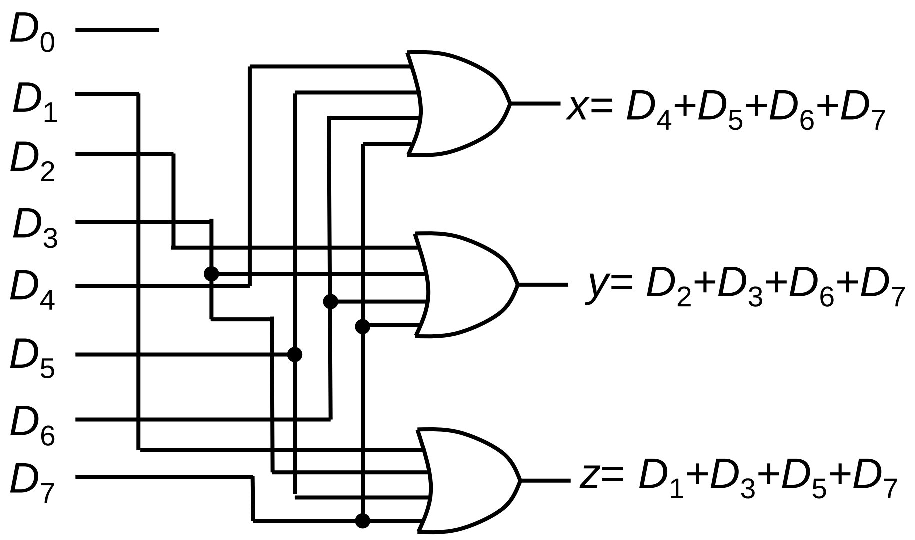
Combinational Logic

Encoder:

- Reverse operation from that of a decoder.
- *m* x *n* encoder: 2^n or less (*m*) input lines and *n* output lines.

Octal to binary encoder.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
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



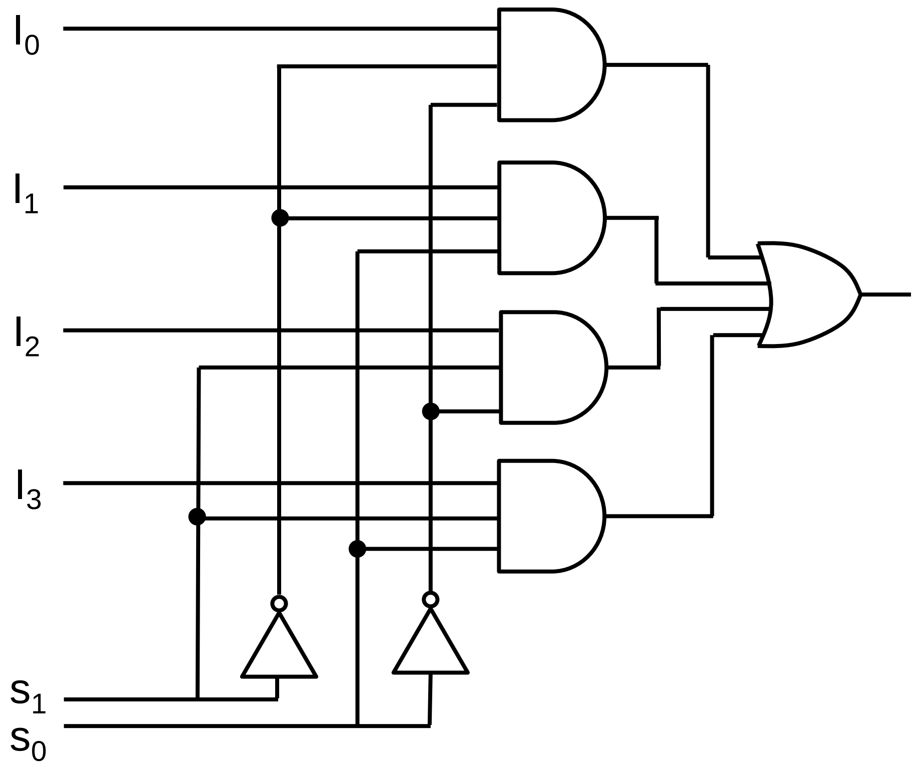
Octal-to-binary encoder.

Combinational Logic

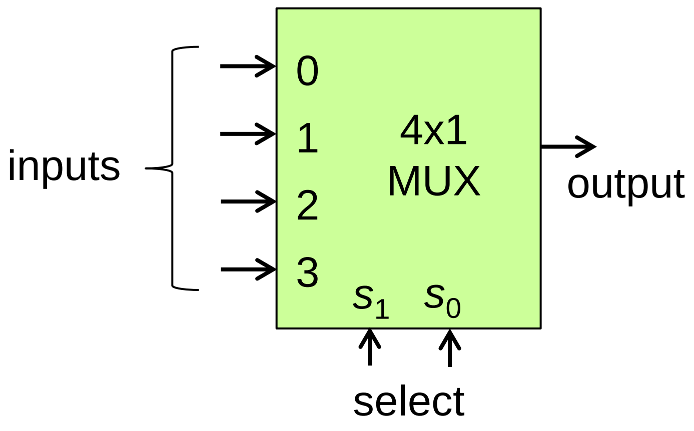
Multiplexers:

Selects binary information from one of many input lines and directs it to a single output line.

Usually, 2^n input lines, n selection lines and one output line.



A 4-to-1 line multiplexer.



A 4-to-1 line multiplexer.

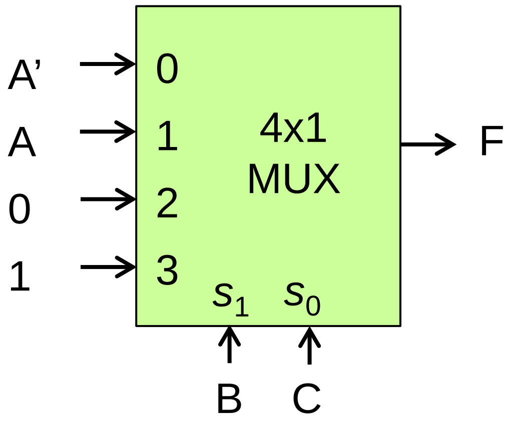
Combinational Logic

Implementation of a function using MUX:

1. Implement the following function using MUX: $F = \sum(0,3,5,7)$.

Implementation table.

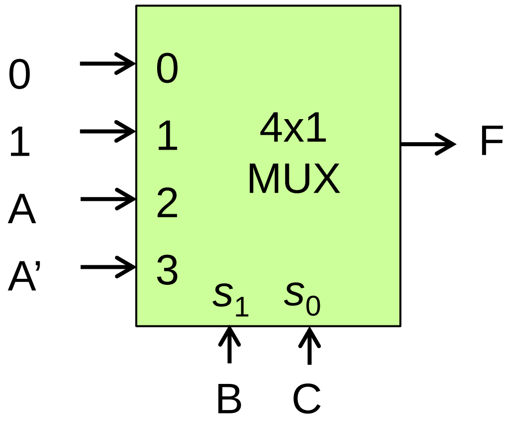
	I_0	I_1	I_2	I_3
A’	$\textcircled{0}$	1	2	$\textcircled{3}$
A	4	$\textcircled{5}$	6	$\textcircled{7}$
	A’	A	0	1



2. Implement the following function using MUX: $F = \sum(1,3,5,6)$.

Implementation table.

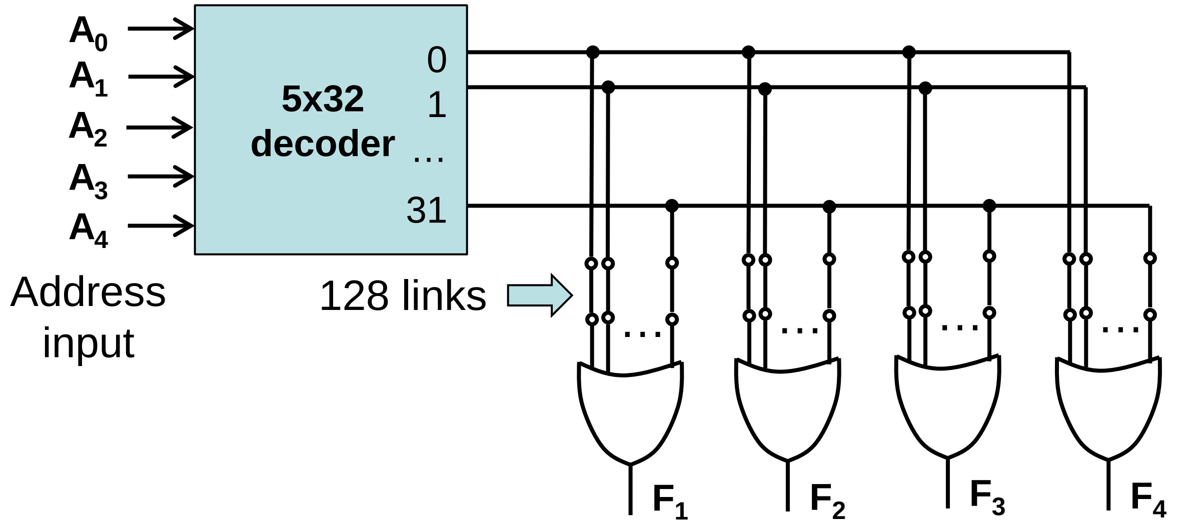
	I_0	I_1	I_2	I_3
A’	0	$\textcircled{1}$	2	$\textcircled{3}$
A	4	$\textcircled{5}$	$\textcircled{6}$	7
	0	1	A	A’



Combinational Logic

Read Only Memory (ROM):

- Programmable memory device that includes both the decoder and OR gates.
- IC package, eliminate all interconnecting wires.
- Internal links can be fused or broken.
- Each bit combination of the input variables (*n*) is called an address.
- Each bit combination that comes out as outputs are called word (2^n).
- Number of bits per word = *m*, then the ROM is called a 2^n x m ROM.



Logic construction of a 32x4 ROM.

Combinational Logic

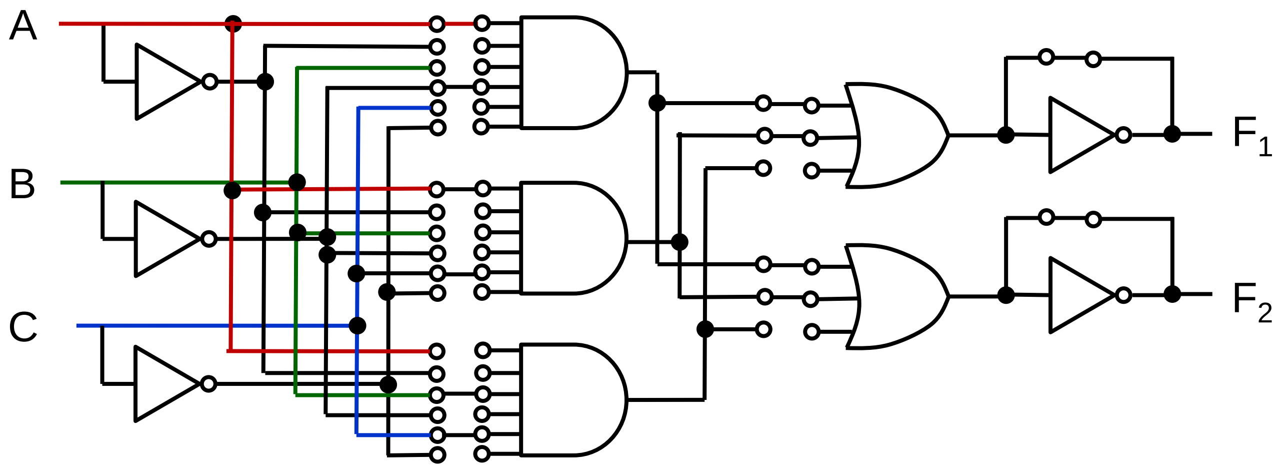
Programmable Logic Array (PLA):

- In a ROM, don't care conditions become address inputs that will never occur.
- A PLA consist of a group of AND gates and OR gates.
- Each of the AND gate can be programmed to generate a product term.

Implement the following functions by using a 3x3x2 (number of input x number of product term x number of output) PLA.

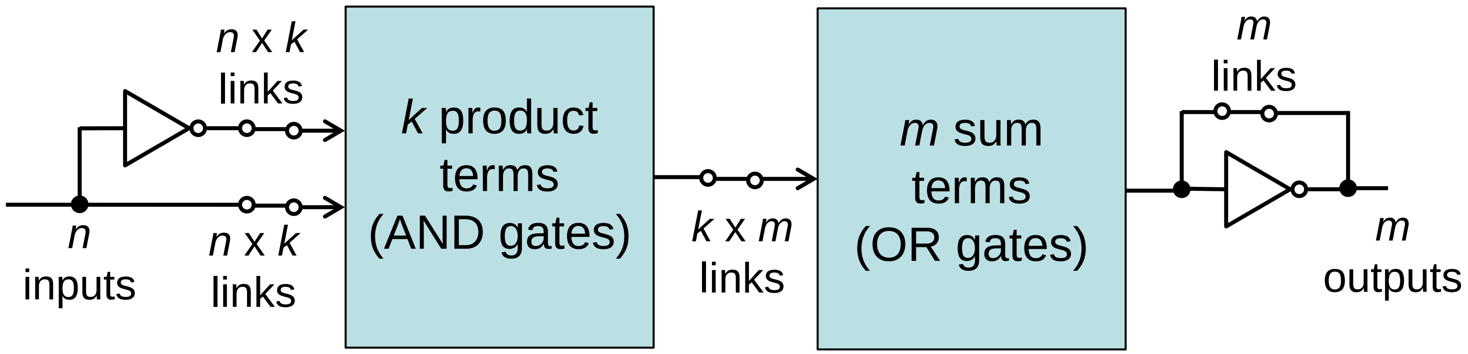
$$F_1 = AB' + AC$$

$$F_2 = AC + BC.$$



Implementation of the functions with PLA.

Combinational Logic



PLA block diagram.

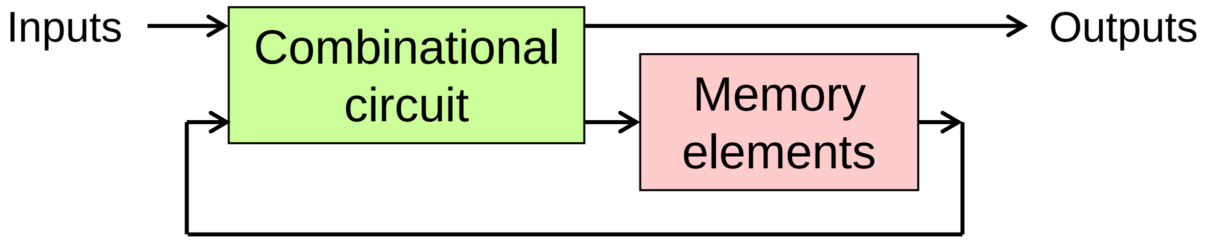
Calculate the total number of links of a 16 x 48 x 8 PLA.

Solution:

Number of links = $2n \times k + k \times m + m$
= 1928.

Number of links for a similar size ROM = $2^n \times m$
= 524288.

Sequential Logic



Block diagram of a sequential circuit.

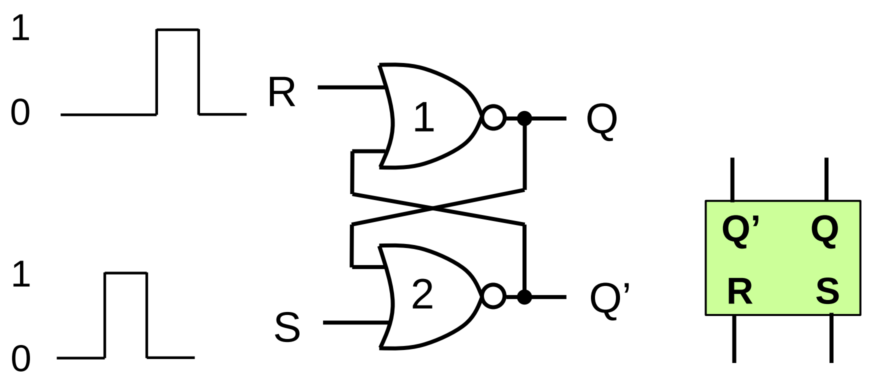
As shown, the outputs depend not only on the external inputs but also on the present state of the memory element.

Depending on the timing of the signal, they are of two types:

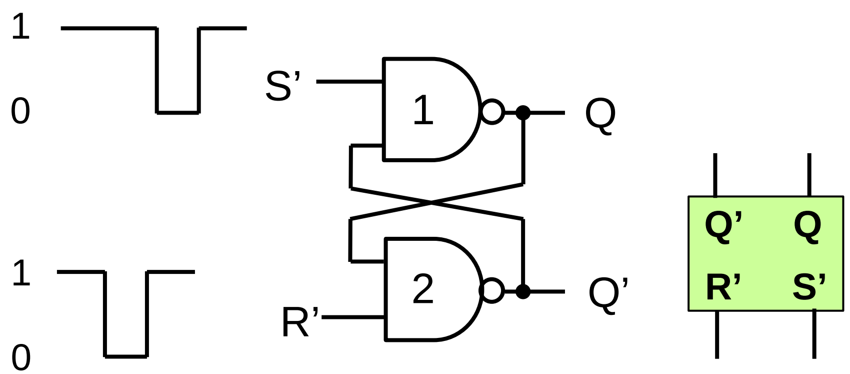
1. Synchronous sequential circuit: behavior is defined only at discrete instant of time. Synchronization is achieved by a master clock generator that generates periodic pulses.
2. Asynchronous sequential circuit: can be affected at any instant of time. Usually, time delay devices are used as a memory device.

Sequential Logic

RS/ SR latch: NOR gate: output is zero if any input is 1.
NAND gate: output is 1 if any input is 0.



SR latch with NOR gates.



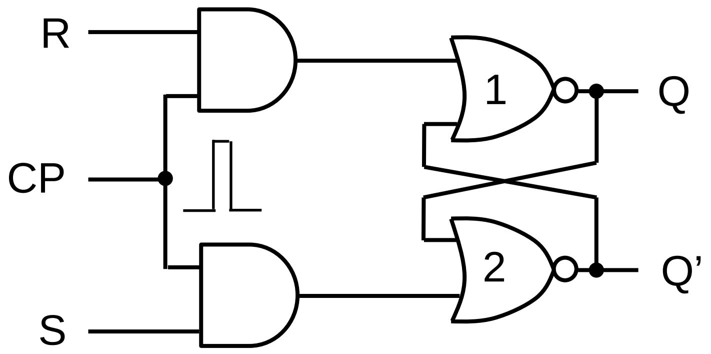
SR latch with NAND gates.

S	*R*	*Q*	*Q'*	
1	0	1	0	
0	0	1	0	After *S* = 1, *R* = 0.
0	1	0	1	
0	0	0	1	After *S* = 0, *R* = 1.
1	1	0	0	Not allowed.

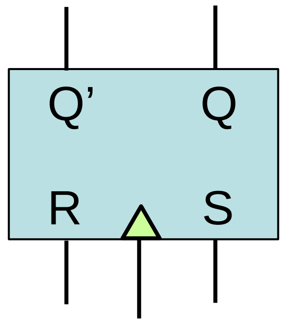
S'	*R'*	*Q*	*Q'*	
1	0	0	1	
1	1	0	1	After *S'* = 1, *R'* = 0.
0	1	1	0	
1	1	1	0	After *S* = 0, *R* = 1.
0	0	1	1	Not allowed.

Sequential Logic

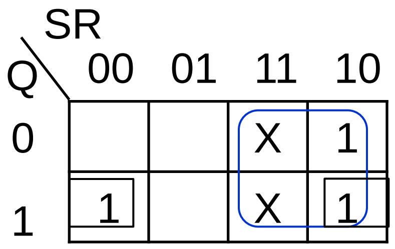
RS Flip-Flop: The basic RS latch with AND gates respond to input levels when the clock pulse is 1: synchronous sequential circuit.



Clocked RS Flip-Flop.



Graphical symbol.



Q \ SR	00	01	11	10
0			X	1
1	1		X	1

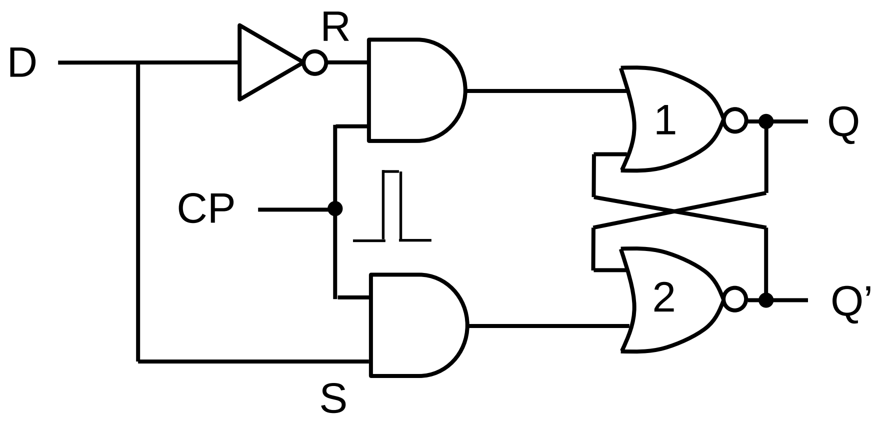
$$Q(t+1) = S + R'Q.$$

Characteristic table.

Q	S	R	Q (t+1)	
0	0	0	0	
0	0	1	0	Reset
0	1	0	1	Set
0	1	1	NA	
1	0	0	1	
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	NA	

Sequential Logic

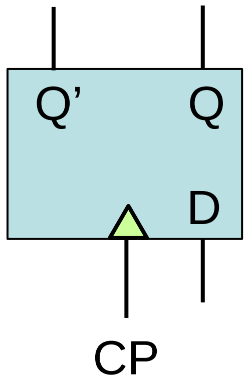
D Flip-Flop: D flip-flop is used to transfer the data.



A gated D Flip-Flop based on RS latch.

Characteristic table.

Q	***D***	***Q* (t+1)**
0	0	0
0	1	1
1	0	0
1	1	1



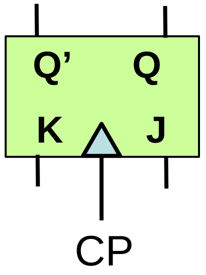
Graphical symbol.

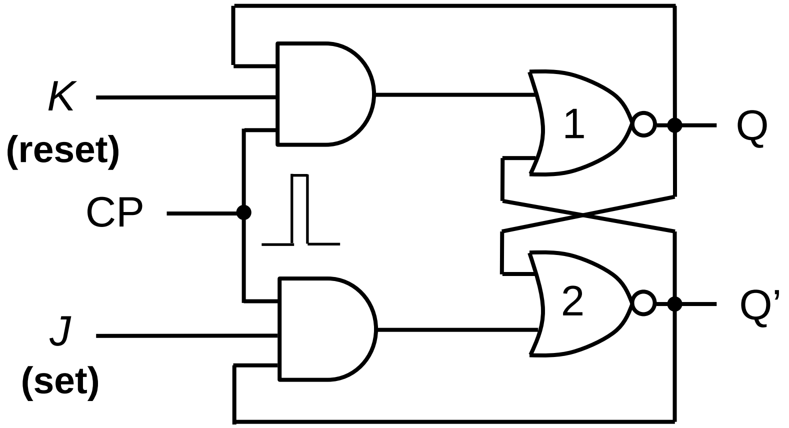
$$\begin{array}{c|cc} Q \backslash D & 0 & 1 \\ \hline 0 & & \boxed{1} \\ 1 & & \boxed{1} \end{array}$$

$$Q(t+1) = D.$$

Sequential Logic

JK Flip-Flop: A refinement of the RS flip-flop by interpreting the S = R = 1 condition as a flip or toggle.
•One problem is that the output for J = 1, K = 1 continue to toggle if the clock pulse is ON for long time.

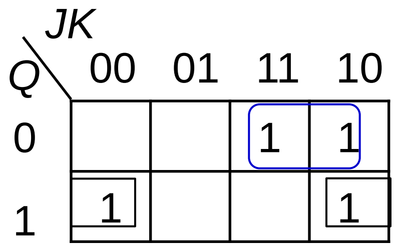




A gated JK flip-flop based on RS latch.

Characteristic table.

Q	J	K	Q (t+1)	
0	0	0	0	
0	0	1	0	
0	1	0	1	Set
0	1	1	1	
1	0	0	1	
1	0	1	0	Reset
1	1	0	1	
1	1	1	0	



Q \ JK	00	01	11	10
0			1	1
1	1			1

$Q(t+1) = JQ' + K'Q.$

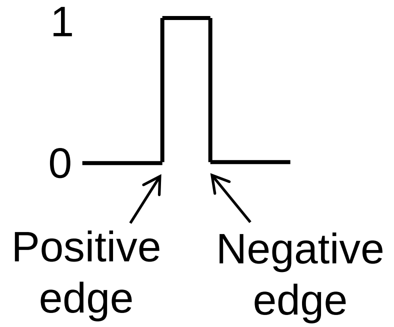
T flip-flop:

- Let K is the only input represented by T and it is also connected to J terminal.
- Then Q changes state for T =1 and remain as it is for T = 0.

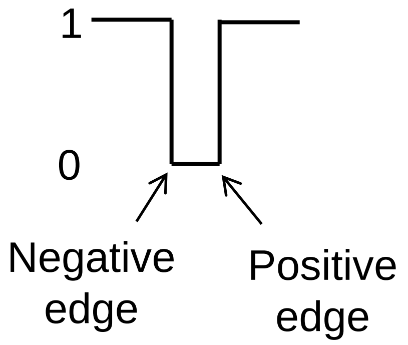
Sequential Logic

Triggering of flip-flop:

- The state of a flip-flop is switched by a momentary change in input signal: triggering.
- For proper operation, a specific relationship must be maintained between the time interval between two changes of input values and the propagation delay.
- For synchronous circuit, propagation delay must be more than the pulse width.
- The above problem can be solved by choosing a circuit that respond only to positive or negative transition /edges e.g. by using capacitor coupling.
- A master-slave flip-flop can avoid this problem.



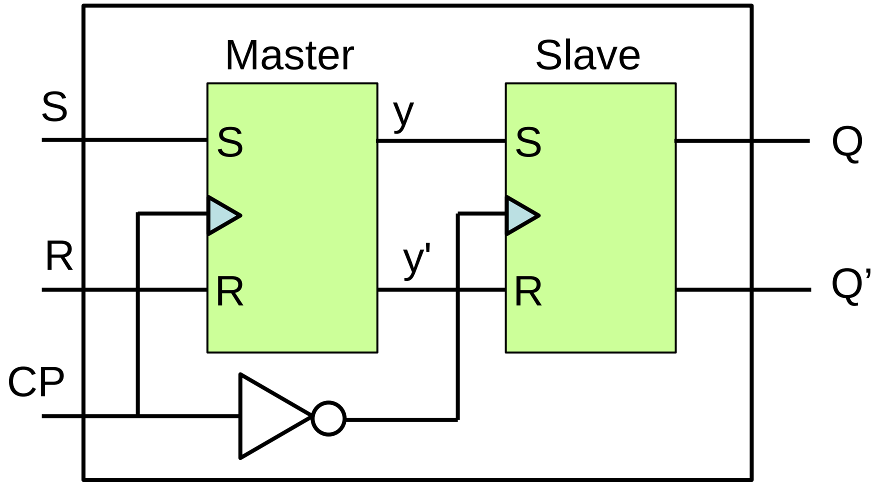
Positive pulse



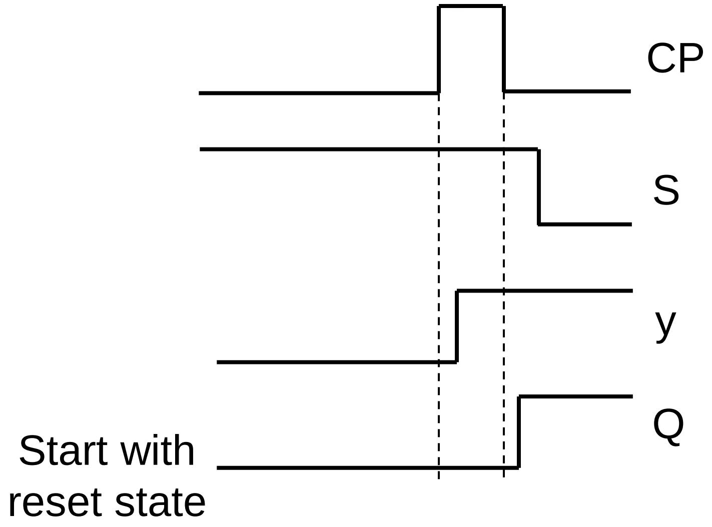
Negative pulse

Sequential Logic

Master Slave flip-flop:



Master-slave flip-flop.



Timing relationship.

- CP = 0: slave is enabled, Q = y, Q' = y' (only 0,1 or 1,0 are possible for the y, y', respectively).
- CP = 1: master is enabled. External R and S are transmitted to the master flip-flop.
- Therefore, at the positive edge of the clock pulse triggers the master, while the negative edge triggers the slave: whole cycle occurs in one period of the pulse.

Questions

1. Implement OR and X-OR gates using minimum number of NMOS.
2. Design a combinational circuit that generates the 9's complement of a BCD digit.
3. Implement the following function by using minimum number of two-input NAND gates. $F = A(B+CD)+BC'$
4. Design a full-adder using minimum number of two-input NOR gates.
5. Design a full-subtractor using minimum number of 4×1 MUX.
6. Implement the following functions by using a 3×8 decoder and additional logic gates if necessary. $F_1 = \sum(0,1,3),\ F_2 = \sum(1,2,3).$
7. Design a BCD-to-decimal decoder that provides an output of all 0's when any invalid input combination occurs.
8. Obtain the characteristics table of a JK′ flip-flop (it is a JK flip-flop with an inverter between external input K′ and internal input K).
9. The carry input z of a full-adder comes from a D flip-flop. Output of the full-adder S provides the sum of the three inputs. The carry is connected to input of the flip-flop. Obtain the characteristic table of the sequential circuit.

Thank you