Three Representations of Logic Functions

	AND	OR	NOT
1. Logic Expression	X.Y	X+Y	X' X ~X !X

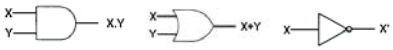
2. Truth Table

X	Y	X.Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

X	X'
0	1
1	0

3. Circuit Diagram / Schematic



Logic Functions of 2 Variables

X Y	F0	F1	F2	F3	F4	F5	F6	F7
0 0	0	0	0	0	0	0	0	0
0 1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1

X Y	F8	F9	F10	F11	F12	F13	F14	F15
0 0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1

- F1 is called a logical AND, denoted by X.Y
- F6 is called an XOR (Exclusive-OR), denoted by $X \oplus Y$
- F7 is called OR, denoted by X + Y
- F8 is NOR, denoted by $\overline{X + Y}$
- F9 is called an XNOR (Exclusive-NOR), denoted by $\overline{X \oplus Y}$
- F14 is NAND, denoted by $\overline{X.Y}$

Truth Tables for 2 Variable Functions

X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

X	Y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

X	Y	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Which Truth Tables Are the Same?

1)

B	A	F
0	0	0
0	1	0
1	0	1
1	1	0

2)

A	B	F
0	0	0
0	1	0
1	0	1
1	1	0

3)

B	A	F
1	1	0
1	0	1
0	1	0
0	0	0

Logic Gate Symbols

- AND denoted by X.Y

- OR denoted by X + Y

- XOR denoted by $X \oplus Y$

- NOT denoted by X' or X

- NAND denoted by $\overline{X.Y}$

- NOR denoted by $\overline{X + Y}$

- XNOR denoted by $\overline{X \oplus Y}$

Half and Full Adder Truth Tables

1)

B	A	S
0	0	0
0	1	1
1	0	1
1	1	0

2)

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

C	B	A	S
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

C	B	A	C
0	0	0	0
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	1

Truth Table with Three Inputs

- Three inputs X, Y, and Z; Output is F
- Logic Function:
  F = 1 if and only if there is a 0 to the left of a 1 in the input
- Truth Table:

X	Y	Z	F	Min term
0	0	0	0	
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	0	
1	0	1	1	
1	1	0	0	
1	1	1	0	

- Logic Expression:
  F=

Truth Table with Four Inputs

- Four inputs X, Y, Z, and W; Output is F
- Logic Function:
  F = 1 if and only if number of variables with value 1 is more than the number of variables with value 0
- Truth Table:
- Logic Expression:
  F=

XYZW	F
0000	0
0001	0
0010	0
0011	0
0100	0
0101	0
0110	0
0111	1
1000	0
1001	0
1010	0
1011	1
1100	0
1101	1
1110	1
1111	1

The Idea of Min Term / Product Term

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	1

A	B
1	0
0	0
0	0
0	1

X	Y	Min Term
0	0	X' Y'
0	1	X' Y
1	0	X Y'
1	1	X Y

- Each row in a truth table represents a unique combination of variables
- Each row can be expressed as a logic combination specifying when that row combination is equal to a 1
- The term is called a MIN TERM or a PRODUCT TERM
- Thus F = A + B = X'Y' + XY

Truth Table with Two Inputs

- Two inputs X and Y; Output is F
- Logic Function:
  F = 1 if and only if X = Y
- Truth Table:

X	Y	F	Min Term
0	0	1	X'Y'
0	1	0	X'Y
1	0	0	XY'
1	1	1	XY

- Logic Expression:
  F=X'Y'.1 + X'Y.0 + XY'.0 + XY.1
  =X'Y' + XY

Min / Product terms for more variables

XYZ	Min Term
000	X' Y' Z'
001	X' Y' Z
010	X' Y Z'
011	X' Y Z
100	X Y' Z'
101	X Y' Z
110	X Y Z'
111	X Y Z

XYZW	Min Term
0000	X' Y' Z' W'
0001	X' Y' Z' W
0010	X' Y' Z W'
0011	X' Y' Z W
0100	X' Y Z' W'
0101	X' Y Z' W
0110	X' Y Z W'
0111	X' Y Z W
1000	X Y' Z' W'
1001	X Y' Z' W
1010	X Y' Z W'
1011	X Y' Z W
1100	X Y Z' W'
1101	X Y Z' W
1110	X Y Z W'
1111	X Y Z W

Multiplexer Circuit

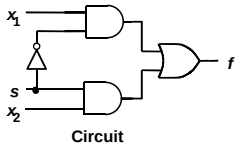
sx_1x_2	$f(s, x_1, x_2)$
000	0
001	0
010	1
011	1
100	0
101	1
110	0
111	1

Truth Table

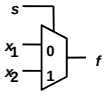
s	$f(s, x_1, x_2)$
0	x_1
1	x_2

More compact truth-table representation

$f = s'.x_1 + s.x_2$



Circuit



Graphical symbol

Canonical Sum-of-Product Expression

- A product (min) term is a unique combination of variables:
  - It has a value of 1 for only one input combination
  - It is 0 for all the other combinations of variables
- To write an expression, we need not write the entire truth table
- We only need those combinations for which function output is 1
- For example, for the function below: f = x'yz'+xy'z'+xyz

x	y	z	f	Min term
0	0	0	0	
0	0	1	0	
0	1	0	1	x'yz'
0	1	1	0	
1	0	0	1	xy'z'
1	0	1	0	
1	1	0	0	
1	1	1	1	xyz

- This is called the Canonical Sum-of-Product (SOP) Expression

Shorthand Notation for Canonical SOP

- We can also assign an integer to represent each input combination
- Thus the function produces a 1 for input combinations 2, 4, 7
- Therefore, the function can be written as f(x,y,z)= $\sum m(2,4,7)$

x	y	z	f	Index for shorthand notation
0	0	0	0	0
0	0	1	0	1
0	1	0	1	2
0	1	1	0	3
1	0	0	1	4
1	0	1	0	5
1	1	0	0	6
1	1	1	1	7

Max / Sum Terms

- A max (sum) term is also a unique combination of variables
  - However, it is opposite of a min term
  - It has a value of 0 for only one input combination
  - It is 1 for all the other combinations of variables
  - That is why it is called a max (sum) term
  - Each row in truth table has a max term corresponding to it
- Example, a max term (x+y+z) is 0 for combination xyz=000 only

X	Y	Max Term
0	0	X + Y
0	1	X + Y'
1	0	X' + Y
1	1	X' + Y'

Canonical Product-of-Sum Expression

- A function can also be written in terms of max terms
- The function is product of all max terms for which function is 0
- For example, the same function of three variable x, y, and z produces 0 for xyz=000, 011, 101, then
  - F = (x+y+z).(x+y'+z').(x'+y+z')
- This is called the Canonical Product-of-Sum (POS) Expression
- The function can also be written as F(x,y,z)= $\prod M\ (0,3,5)$

X	Y	Z	F	F'
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0

F = (F')'
= (x'y'z' + x'yz + xy'z)'
= (x+y+z).(x+y'+z').(x'+y+z')

Truth Tables and Logic Expression for Adder

A	B	C	X	Y
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$X(A,B,C) = \sum m(1,2,4,7)$

$Y(A,B,C) = \sum m(3,5,6,7)$

$X(A,B,C) = \prod M(\quad)$

$Y(A,B,C) = \prod M(\quad)$

X = A'B'C + A'BC' + AB'C' + ABC

Y = A'BC + AB'C + ABC' + ABC

Multiple Forms and Equivalence

- Canonical Sum-of-Product form
- Canonical Product-of-sum form
- How to convert one from other?
- Minterm expansion of X to minterm expansion of X'
  - Just take the terms that are missing

$X(A,B,C) = \sum m(1,2,4,7)$ $\quad X'(A,B,C) = \sum m(\quad)$

- Maxterm expansion of X to maxterm expansion of X'
  - Just take the terms that are missing

$X(A,B,C) = \prod M(\quad)$ $\quad X'(A,B,C) = \prod M(\quad)$

Boolean Algebra

George Boole
1815-1864

- An algebraic structure consists of
  - a set of elements {0, 1}
  - binary operators {+, .}
  - and a unary operator { ' }
- Introduced by George Boole in 1854
- An effective means of describing circuits built with switches
- A powerful tool that can be used for designing and analyzing logic circuits

Axioms of Boolean Algebra

- 1a: 0.0 = 0
  1b: 1+1 = 1
- 2a: 1.1 = 1
  2b: 0+0 = 0
- 3a: 0.1 = 1.0 = 0
  3b: 1+0 = 0+1 = 1
- 4a: If x=0, then x' = 1
  4b: If x=1, then x' = 0

Single-Variable Theorems

- 5a: x.0 = 0 — Null
  5b: x+1 = 1
- 6a: x.1 = x — Identity
  6b: x+0 = x
- 7a: x.x = x — Idempotency
  7b: x+x = x
- 8a: x.x' = 0 — Complementarity
  8b: x+x' = 1
- 9: (x')' = x — Involution

Two- and Three-Variable Properties

- 10a: x.y = y.x — Commutative
  10b: x+y = y+x
- 11a: x.(y.z) = (x.y).z — Associative
  11b: x+(y+z) = (x+y)+z
- 12a: x.(y+z) = x.y + x.z — Distributive
  12b: x+y.z = (x+y).(x+z)
- 13a: x+x.y = x — Absorption
  13b: x.(x+y) = x

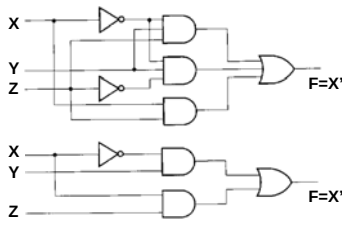
Other Properties

- Combining
  14a: x.y+x.y' = x
  14b: (x+y).(x+y') = x
- DeMorgan's Theorem
  15a: (x.y)' = x'+y'
  15b: (x+y)' = x'.y'
- Another form of Absorption
  16a: x+x'.y = x+y
  16b: x.(x'+y) = x.y

Simplify Logic Function by Algebraic Manipulation

- F =X'YZ+X'YZ'+XZ
  =X'Y(Z+Z')+XZ — by Distributive
  =X'Y(1)+XZ — by Complementarity
  =X'Y+XZ — by Identity

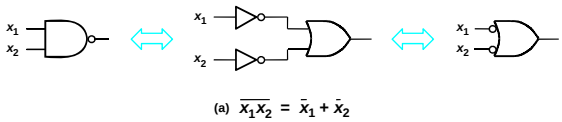
X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



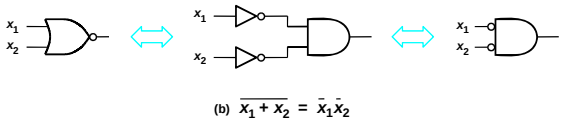
Principle of Duality

- **Dual:**
  - **A dual of a Boolean expression is derived by replacing . by +, + by ., 0 by 1, and 1 by 0 and leaving variables unchanged**
  - **In general duality: $f^D(x1,x2,\ldots,xn,0,1,+,.) = f(x1,x2,\ldots,xn,1,0,.,+)$**
- **Principle of Duality:**
  - **If any theorem can be proven, the dual theorem can also be proven.**
  - **A meta-theorem (a theorem about theorems)**
- **Examples:**
  - **Multiplication and factoring:**
    - $(x+y).(x'+z) = x.z+x'.y$ and $x.y+x'.z=(x+z).(x'+y)$
  - **Consensus:**
    - $(x.y)+(y.z)+(x'.z)=x.y+x'z$ and

DeMorgan's Theorem in Terms of Logic Gates

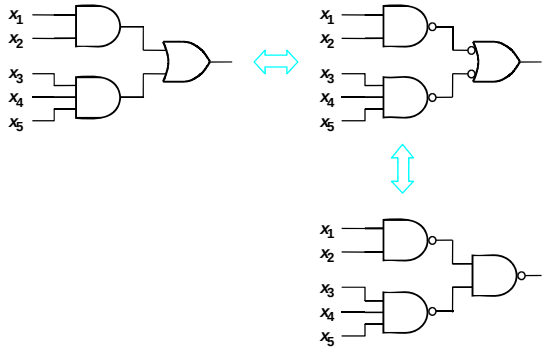


(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$

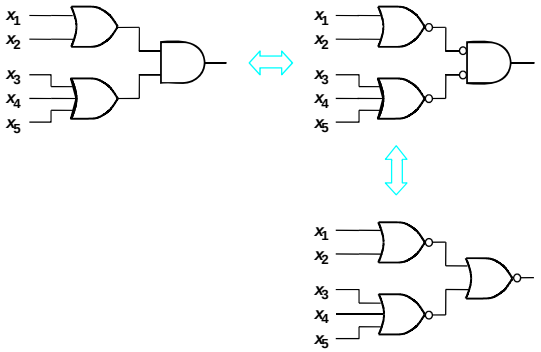


(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Using NAND to Implement SOP



Using NOR to Implement POS



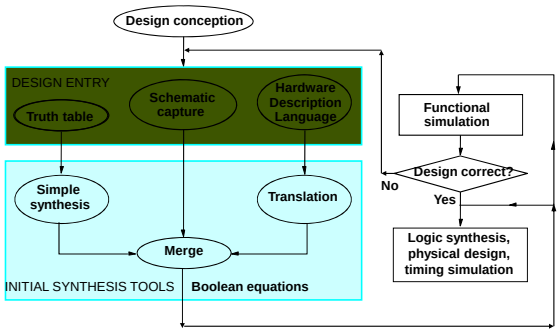
Order of Precedence of Logic Operators

- **From highest precedence to lowest: NOT, AND, OR**
- **We can use parenthesis to change the order**

- **Examples:**
  f = X'+X.Y is the same as
  f = ((X')+(X.Y))

  f = X.(Y+Z) is NOT the same as
  f = X.Y+Z

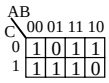
A Typical CAD (Computer-Aided Design) System



Karnaugh map

- Karnaugh map (K-map) allows viewing the function in a picture form
- Containing the same information as a truth table
- But terms are arranged such that two neighbors differ in only one variable
- It is easy to identify which terms can be combined
- Example:

A map with 3 variables

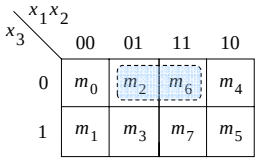


A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Location of Min-terms in K-maps

x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

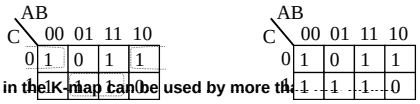


(b) Karnaugh map

$\mathbf{m_2 + m_6 = x_1' x_2 x_3' + x_1 x_2 x_3'}$
$\mathbf{= x_2 x_3'}$

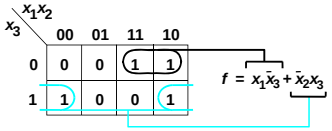
Simplification using K-map

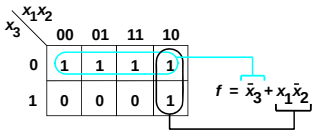
- Groups of '1's of size 1x1, 2x1, 1x2, 2x2, 4x1, 1x4, 4x2, 2x4, or 4x4 are called prime implicants (p.159 in textbook).



- A '1' in the K-map can be used by more than one group.
- Some rule-of-thumb in selecting groups:
  - Try to use as few group as possible to cover all '1's.
  - For each group, try to make it as large as you can (i.e., if you can use a 2x2, don't use a 2x1 even if that is enough).

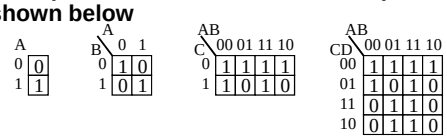
Examples of 3-Variable K-map





Karnaugh maps with up to 4 variables

- **Example: 1, 2, 3, and 4 variables maps are shown below**



- **What if a function has 5 variables?**

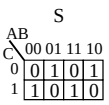
K-map Example for Adder functions

A	B	C	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

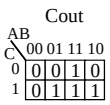
$S(A,B,C) = \sum m(1,2,4,7)$

$Cout(A,B,C) = \sum m(3,5,6,7)$

S



Cout



S = A'B'C + A'BC' + AB'C' + ABC

Cout = BC + AC + AB

K-map with Don't Care Conditions

- ***Don't care condition*** **is input combination that will never occur.**
- **So the corresponding output can either be 0 or 1.**
- **This can be used to help simplifying logic functions.**
- **Example: F(A,B,C,D)=Σ m(1,3,7,11,15)+ Σ D(0,2,5)**

AB \ CD	00	01	11	10
00	d	1	1	d
01	0	d	1	0
11	0	0	1	0
10	0	0	1	0

F = CD+A'B'

AB \ CD	00	01	11	10
00	d	1	1	d
01	0	d	1	0
11	0	0	1	0
10	0	0	1	0

F = CD+A'D

d: Don't Care Condition

Examples

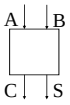
- **Simplify the following function considering:**
  - **the sum-of-products form**
  - **-- the product-of-sums form**

AB \ CD	00	01	11	10
00	1	0	0	1
01	1	1	1	d
11	0	d	1	1
10	0	0	0	1

AB \ CD	00	01	11	10
00	1	0	0	1
01	1	1	1	d
11	0	d	1	1
10	0	0	0	1

1-bit building blocks to make n-bit circuit

- **Design a 1-bit circuit with proper "glue logic" to use it for n-bits**
  - **It is called a bit slice**
  - **The basic idea of bit slicing is to design a 1-bit circuit and then piece together n of these to get an n-bit component**
- **Example:**
- **A half-adder adds two 1-bit inputs**
- **Two half adders can be used to add 3 bits**
- **A 3-bit adder is a full adder**
- **A full adder can be a bit slice to construct an n-bit adder**

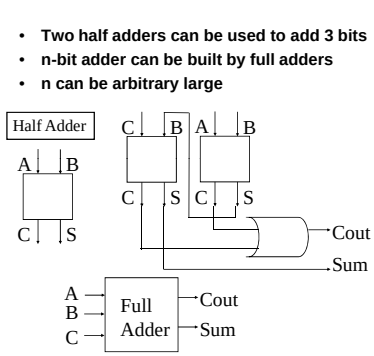


A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Adder

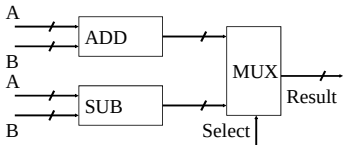
Full adder & multi-bit ripple-carry adder

- **Two half adders can be used to add 3 bits**
- **n-bit adder can be built by full adders**
- **n can be arbitrary large**



Multiple Function Unit Design

- **Design a unit that can do more than one function**
- **In that case, we can design a function unit for each operation like ADD, SUB, AND, OR, ...**
- **And then select the desired output**
- **For example, if we want to be able to perform ADD and SUB on two given operands A and B, and select any one**
- **Then the following set up will work**



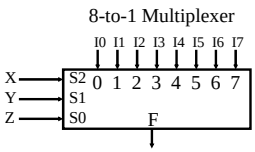
ADD/SUB unit design

- **Separate ADD and SUB units are expensive**
- **We can simplify the design**
- **A - B is the same as adding negation of B to A**
- **How to negate?**
  - **2's complement (i.e., take 1's complement and add 1)**
  - **Adding 1 is also expensive**
  - **It needs an n-bit adder in general**
  - **However, we only need to add two bits in each stage**
    - In the first stage, we need to add 1's complement of LSB and 1
    - In other stages, we need to add carry output of previous bit to 1's complement of current bit
- **We select B or negation of B depending on the requirement**
- **We add A to the selected input to obtain the result**

Multiplexing and Multiplexer

- Multiplexers are circuits which select one of many inputs
- In here, we assume that we have one-bit inputs (in general, each input may have more than one bit)

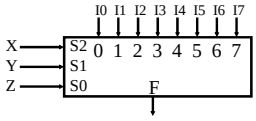
- Suppose we have eight inputs: I0, I1, I2, I3, I4, I5, I6, I7
- We want one of them to be output based on selection signals
- 3 bits of selection signals to decide which input goes to output
- Note the order of selection signals
  - X is MSB and Z is LSB



X	Y	Z	F
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

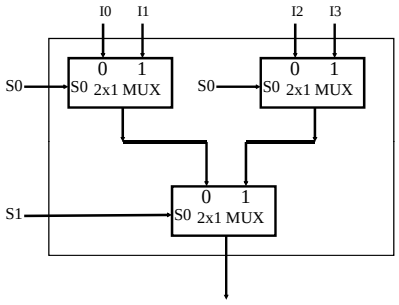
Multiplexer Implementation

- We can write a logic expression for output F as follows
  F = X' Y' Z' I0 + X' Y' Z I1 + X' Y Z' I2 + X' Y Z I3
  + X Y' Z' I4 + X Y' Z I5 + X Y Z' I6 + X Y Z I7
- This circuit can be implemented using
  - eight 4-input AND gates and one 8-input OR gates



X	Y	Z	F
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

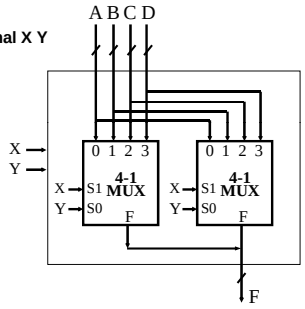
Implementing 4-to-1 MUX using 2-to-1 MUXs



Making a 2-bit 4-to-1 Multiplexer

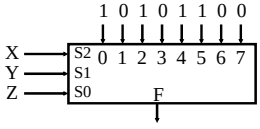
- Four *2-bit* inputs A, B, C, D
- One *2-bit* output F
- Two bits of selection signal X Y

X	Y	F
0	0	A
0	1	B
1	0	C
1	1	D



Synthesis of Logic Functions using Multiplexers

- Multiplexers can be directly used to implement a function
- Easiest way is to use function inputs as selection signals
- Input to multiplexer is a set of 1s and 0s depending on the function to be implemented
- We use a 8-to-1 multiplexer to implement function F
- Three select signals are X, Y, and Z, and output is F
- Eight inputs to multiplexer are 1 0 1 0 1 1 0 0
- Depending on the input signals
  - multiplexer will select proper output

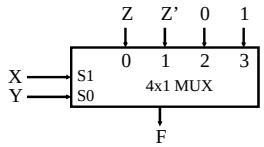


X	Y	Z	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Implementing 3-variable functions with 4x1 MUX

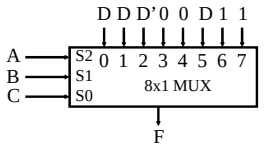
- Divide the outputs into 4 groups based on X and Y.
- Write the outputs as a function of Z
- There are only 4 possibilities: F=Z, F=Z', F=0, F=1

X	Y	Z	F	
0	0	0	0	F=Z
0	0	1	1	
0	1	0	1	F=Z'
0	1	1	0	
1	0	0	0	F=0
1	0	1	0	
1	1	0	1	F=1
1	1	1	1	



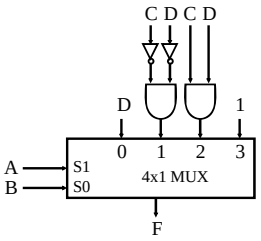
Implementing 4-variable functions with 8x1 MUX

A	B	C	D	F	
0	0	0	0	0	F=D
0	0	0	1	1	
0	0	1	0	0	F=D
0	0	1	1	1	
0	1	0	0	1	F=D'
0	1	0	1	0	
0	1	1	0	0	F=0
0	1	1	1	0	
1	0	0	0	0	F=0
1	0	0	1	0	
1	0	1	0	0	F=D
1	0	1	1	1	
1	1	0	0	1	F=1
1	1	0	1	1	
1	1	1	0	1	F=1
1	1	1	1	1	



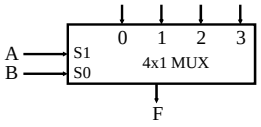
Implementing 4-variable functions with 4x1 MUX

A	B	C	D	F	
0	0	0	0	0	F=D
0	0	0	1	1	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	1	F=C'D'
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	0	F=CD
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	1	
1	1	0	0	1	F=1
1	1	0	1	1	
1	1	1	0	1	
1	1	1	1	1	



Implementing 4-variable functions with 4x1 MUX

A	B	C	D	F	
0	0	0	0	0	F=
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	0	
0	1	0	0	0	F=
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	1	
1	0	0	0	0	F=
1	0	0	1	0	
1	0	1	0	1	
1	0	1	1	1	
1	1	0	0	0	F=
1	1	0	1	1	
1	1	1	0	1	
1	1	1	1	1	

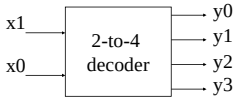


Definition of Decoder

- Suppose we have n input bits (which can represent up to 2^n distinct elements of coded information).
- We need a device that allows us to select which of the 2^n elements, devices, memory locations, etc. is being selected.
- In general:
  - A decoder has n input bits
  - A decoder has 2^n (or less) output bits
  - As a rule, all but one of the outputs is zero (deselected) at any time (called *one-hot encoded*)

2-to-4 Decoder

- The 2-to-4 decoder is a block which decodes the 2-bit binary inputs and produces four outputs
- One output corresponding to the input combination is a one
- Two inputs and four outputs are shown in the figure
- The equations are
  - y0 = x1'. x0'
  - y1 = x1'. x0
  - y2 = x1 . x0'
  - y3 = x1 . x0
- The truth table:



x1	x0	y3	y2	y1	y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

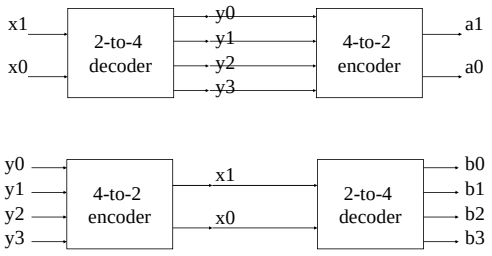
Definition of Encoder

- Encoders perform the inverse function of Decoders.
- An encoder has 2^n (or less) input bits and n output bits
- The output bits generate the binary code corresponding to the input value
- Assuming only one input has a value of 1 at any given time
- Example: An 8-to-3 Encoder

Inputs								Outputs		
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

A2=D4+D5+D6+D7
A1=D2+D3+D6+D7
A0=D1+D3+D5+D7

What are the outputs of the following circuits?



Priority Encoders

- Each input signal has a priority level associated with it
- May have more than one 1's in the input signals
- Outputs indicate the active input that has the highest priority
- Example: 4-to-2 priority encoder
  - Assume w3 has the highest priority and w0 the lowest
  - y1 y0 indicate the active input with highest priority
  - z indicates none of the inputs is equal to 1

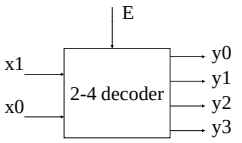
w3	w2	w1	w0	y1	y0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

x: both 0 and 1 (irrelevant)

Let i0 = w0 w1' w2' w3'
i1 = w1 w2' w3'
i2 = w2 w3'
i3 = w3
Then y0 = i1 + i3
y1 = i2 + i3

Decoder with Enable

- A 2-to-4 decoder can be designed with an enable signal
- If enable is zero, all outputs are zero
- If enable is 1, then an output corresponding to two inputs is a one, all others are still zero
- The equations are
  - y0 = x1'. x0'. E
  - y1 = x1'. x0 . E
  - y2 = x1 . x0'. E
  - y3 = x1 . x0 . E

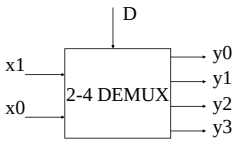


Truth Table for 2-to-4 Decoder with Enable

x1	x0	E	y3	y2	y1	y0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	1	1	0	0	0

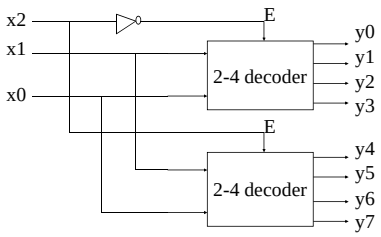
Demultiplexers

- Perform the opposite function of multiplexers
- Placing the value of a single data input onto one of the multiple data outputs
- Same implementation as decoder with enable
- Enable input of decoder serves as the data input for the demultiplexer



3-to-8 decoder using a 2-to-4 decoder with Enable

- The 3-to-8 decoder can be implemented using two 2-to-4 decoders with enable and one NOT gate
- The implementation is as shown



Verilog HDL

Popular Hardware Description Languages (HDLs):

- **Verilog HDL**
  - **More popular with US companies**
  - **Similar to C / Pascal programming language in syntax**
- **VHDL**
  - **More popular with European companies**
  - **Similar to Ada programming language in syntax**
  - **More "verbose" than Verilog**

Uses of Verilog:

- **Synthesis**
- **Simulation**
- **Verification**

Verilog Syntax

- **Module / Signal names:**
  - **Start with a letter**
  - **Follow by any sequence of letter, number, _ and $**
  - **Case sensitive**
- **Comment by // or /* */**
- **White spaces (SPACE, TAB, blank line) are ignored.**

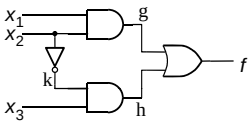
```
// An example
module example1 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;

        and (g, x1, x2);
        not (k, x2);
        and (h, k, x3);
        or (f, g, h);

endmodule
```

```
// An example
module example1 (x1, x2, x3
, f);input x1, x2, x3;output
f;and (g, x1, x2);not (k, x2
);and (h, k, x3); or (f, g,
h);endmodule
```


Structural Specification of Logic Circuit

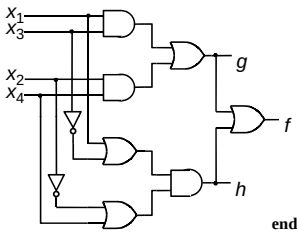



```
module example1 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;

        and (g, x1, x2);
        not (k, x2);
        and (h, k, x3);
        or (f, g, h);

endmodule
```


Another Example of Structural Specification

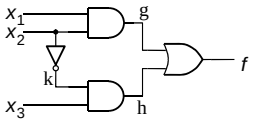



```
module example2 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        and (z1, x1, x3);
        and (z2, x2, x4);
        or (g, z1, z2);
        or (z3, x1, ~x3);
        or (z4, ~x2, x4);
        and (h, z3, z4);
        or (f, g, h);

endmodule
```


Behavioral Specification Continuous Assignment




```
// Structural Specification
module example1 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;

        and (g, x1, x2);
        not (k, x2);
        and (h, k, x3);
        or (f, g, h);

endmodule
```

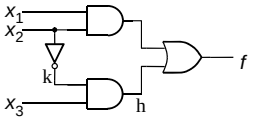
```
// Behavioral Specification
module example3 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;

        assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

Concurrent statement

Behavioral Specification Procedural (Sequential) Statement




```
module example5 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    reg f;

    always @(x1 or x2 or x3)
        if (x2 == 1)
            f = x1;
        else
            f = x3;

endmodule
```

Declared as variable (register) if assigned a value in a procedural statement

Sensitivity list

always *block*