### Three Representations of Logic Functions

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Logic Expression</td>
<td>X.Y</td>
<td>X+Y</td>
</tr>
<tr>
<td>2. Truth Table</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3. Circuit Diagram</td>
<td>/ Schematic</td>
<td></td>
</tr>
</tbody>
</table>

### Logic Functions of 2 Variables

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

### Truth Tables for 2 Variable Functions

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>AND</th>
<th>X</th>
<th>Y</th>
<th>OR</th>
<th>X</th>
<th>Y</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>NAND</th>
<th>X</th>
<th>Y</th>
<th>NOR</th>
<th>X</th>
<th>Y</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Which Truth Tables Are the Same?

<table>
<thead>
<tr>
<th>1)</th>
<th>B</th>
<th>A</th>
<th>F</th>
<th>2)</th>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Logic Gate Symbols

- AND denoted by X.Y
- OR denoted by X + Y
- XOR denoted by X ⊕ Y
- NOT denoted by X' or X
- NAND denoted by X.Y
- NOR denoted by X + Y
- XNOR denoted by X ⊕ Y

### Half and Full Adder Truth Tables

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>S</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</th>
<th>S</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Truth Table with Three Inputs

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
<th>Min Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
- Logic Expression:
  \[ F = \]  

### Truth Table with Four Inputs

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

<table>
<thead>
<tr>
<th>XYZW</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>
- Logic Expression:
  \[ F = \]  

### The Idea of Min Term / Product Term

- 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 1
- The term is called a MIN TERM or a PRODUCT TERM
- Thus F = A + B = X'Y' + XY

### Min / Product terms for more variables

<table>
<thead>
<tr>
<th>XYZ</th>
<th>Min Term</th>
<th>XYZW</th>
<th>Min Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>X'Y'Z'</td>
<td>0000</td>
<td>X'Y'Z'W'</td>
</tr>
<tr>
<td>001</td>
<td>X'Y'Z</td>
<td>0001</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td>010</td>
<td>X'Y'Z'</td>
<td>0010</td>
<td>X'Y'Z'W'</td>
</tr>
<tr>
<td>011</td>
<td>X'Y'Z</td>
<td>0100</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td>100</td>
<td>X'Y'Z'</td>
<td>0101</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td>101</td>
<td>X'Y'Z</td>
<td>0110</td>
<td>X'Y'Z'W'</td>
</tr>
<tr>
<td>110</td>
<td>X'Y'Z</td>
<td>0111</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td>111</td>
<td>X'Y'Z</td>
<td>1000</td>
<td>X'Y'Z'W'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010</td>
<td>X'Y'Z'W'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100</td>
<td>X'Y'Z'W'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1101</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110</td>
<td>X'Y'ZW'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111</td>
<td>X'Y'ZW'</td>
</tr>
</tbody>
</table>

### Truth Table with Two Inputs

- Two inputs X and Y; Output is F
- Logic Function: 
  \[ F = 1 \text{ if and only if } X = Y \]
- Truth Table:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F</th>
<th>Min Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X'Y'</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>XY</td>
</tr>
</tbody>
</table>
- Logic Expression:
  \[ F = X'Y' + XY \]

### Multiplexer Circuit

- More compact truth-table representation
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:
\[ f = x'y'z' + xy'z' + xyz \]

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

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

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
\[ F(x, y, z) = \prod \left( M(0, 3, 5) \right) \]

Truth Tables and Logic Expression for Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ X(A, B, C) = \sum m(1, 2, 4, 7) \]
\[ Y(A, B, C) = \sum m(3, 5, 6, 7) \]

\[ X(A, B, C) = \prod M(\_\_\_) \]
\[ Y(A, B, C) = \prod M(\_\_\_) \]

\[ X = A'B'C + A'BC' + AB'C + ABC \]
\[ Y = A'BC + AB'C + ABC' + ABC \]
**Boolean Algebra**

- An algebraic structure consists of
  - a set of elements \( \{0, 1\} \)
  - binary operators \(+, \cdot\)
  - 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

George Boole
1815-1864

---

**Axioms of Boolean Algebra**

- 1a: \(0 \cdot 0 = 0\)
- 1b: \(1 + 1 = 1\)
- 2a: \(1 \cdot 1 = 1\)
- 2b: \(0 + 0 = 0\)
- 3a: \(0 \cdot 1 = 1 \cdot 0 = 0\)
- 3b: \(1 + 0 = 0 + 1 = 1\)
- 4a: If \(x = 0\), then \(x' = 1\)
- 4b: If \(x = 1\), then \(x' = 0\)

---

**Single-Variable Theorems**

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

---

**Two- and Three-Variable Properties**

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

---

**Other Properties**

- Combining
  - 14a: \(x \cdot y + x \cdot y' = x\)
  - 14b: \((x+y) \cdot (x+y') = x\)

- DeMorgan's Theorem
  - 15a: \((x \cdot y)' = x' + y'\)
  - 15b: \((x+y)' = x' \cdot y'\)

- Another form of Absorption
  - 16a: \(x \cdot x' \cdot y = x + y\)
  - 16b: \(x \cdot (x' + y) = x \cdot y\)

---

**Simplify Logic Function by Algebraic Manipulation**

- \(F = X'YZ + X'YZ' + XZ\)
- \(F = X'Y + XZ\)

---

**X Y Z F**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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: $f(x_1,x_2,...,x_n,0,1,+,.) = f(x_1,x_2,...,x_n,1,0,.,+)$
- Principle of Duality:
  - If any theorem can be proven, the dual theorem can also be proven.
- 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

- $\overline{x_1x_2} = \overline{x_1} + \overline{x_2}$
- $\overline{x_1 + x_2} = x_1x_2$

Using NAND to Implement SOP

```
  x_1
  x_2
  x_3
  x_4
  x_5
  \overline{x_1 + x_2 + x_3 + x_4 + x_5}
```

Using NOR to Implement POS

```
  x_1
  x_2
  x_3
  x_4
  x_5
  \overline{x_1 + x_2 + x_3 + x_4 + x_5}
```

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

```
Design conception
  ▸ Design correctness?
    ▼ Logic synthesis, physical design, timing simulation
    Yes
    ▼ Functional simulation
      ▼ Simple synthesis
        ▼ Translation
          ▼ Merge
            ▼ Boolean equations
```

FUNCTIONAL SIMULATION

- Design correct?
  - No
    - Initial synthesis tools
  - Yes
    - Design correctness

Schematic capture

- Hardware description language

INITIAL SYNTHESIS TOOLS

- Truth table
- Hardware description language

Schematic capture

- Design conception
- Simple synthesis
- Translation
- Merge
- Boolean equations

DESIGN ENTRY

- Design conception
- Truth table
- Hardware description language
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:

\[
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

Location of Min-terms in K-maps

- Example: 1, 2, 3, and 4 variables maps are shown below
- What if a function has 5 variables?

Simplification using K-map

- Groups of ‘1’s of size 1x1, 2x1, 2x2, 4x1, 4x2, 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

- Example: 1, 2, 3, and 4 variables maps are shown below
- What if a function has 5 variables?
**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: $F(A,B,C,D)=\sum m(1,3,7,11,15)+\sum D(0,2,5)$

**Examples**

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

**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

**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
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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>I0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>I1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>I3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>I4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>I5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>I6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>I7</td>
</tr>
</tbody>
</table>

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 gate.

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

Divide the outputs into 4 groups based on X and Y.

Write the outputs as a function of Z.

There are only 4 possibilities: \( F = Z \), \( F = Z' \), \( F = 0 \), \( F = 1 \).

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: \( F = Z \), \( F = Z' \), \( F = 0 \), \( F = 1 \).
Implementing 4-variable functions with 8x1 MUX

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

F = D

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)

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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>D7 D6 D5 D4 D3 D2 D1 D0</td>
<td>A2 A1 A0</td>
</tr>
<tr>
<td>0 0 0 0 0 1 1 0</td>
<td>0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>0 0 0 1 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>x1</th>
<th>x0</th>
<th>y3</th>
<th>y2</th>
<th>y1</th>
<th>y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4x1 MUX
What are the outputs of the following circuits?

2-to-4 decoder
\[ x_1 \quad y_0 \quad y_1 \quad y_2 \quad y_3 \]

4-to-2 encoder
\[ a_1 \quad a_0 \]

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

<table>
<thead>
<tr>
<th>w3</th>
<th>w2</th>
<th>w1</th>
<th>w0</th>
<th>y1</th>
<th>y0</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Let \( i_0 = w_0 w_1' w_2' w_3' \)
\( i_1 = w_1 w_2' w_3' \)
\( i_2 = w_2 w_3' \)
\( i_3 = w_3 \)
Then \( y_0 = i_1 + i_3 \)
\( y_1 = i_2 + i_3 \)

x: both 0 and 1 (irrelevant)

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
  - \( y_0 = x_1'. x_0'. E \)
  - \( y_1 = x_1'. x_0 . E \)
  - \( y_2 = x_1 . x_0'. E \)
  - \( y_3 = x_1 . x_0 . E \)

Truth Table for 2-to-4 Decoder with Enable

<table>
<thead>
<tr>
<th>x1</th>
<th>x0</th>
<th>E</th>
<th>y3</th>
<th>y2</th>
<th>y1</th>
<th>y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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.

Verilog Syntax 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 example2 (x1, x2, x3, f);
input x1, x2, x3;
output f;
assign f = (x1 & x2) | (~x2 & x3);
endmodule

Behavioral Specification Procedural (Sequential) Statement

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

// Behavioral Specification
module example2 (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

Concurrent statement

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

Sensitivity list

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