5.2 Applications

Notebook:	Discrete Mathematics [CM1020]		
Created:	2019-10-07 2:31 PM	**Updated:**	2019-11-23 6:13 PM
Author:	SUKHJIT MANN		

Cornell Notes	**Topic:** 5.2 Applications	Course: BSc Computer Science
		Class: Discrete Mathematics-Lecture
		Date: November 23, 2019

Essential Question:

What is Boolean Algebra, its postulates and/or Boolean functions?

Questions/Cues:

- What is logic gate?
- What is AND, OR, NOT Gate?
- What are the common Gates in circuit design?
- Which of the gate is commutative and/or associative?
- How are De Morgan's laws represented in gates?
- What is a circuit?
- What is meant by building a circuit from a Boolean expression?
- What are the steps to write a Boolean expression from a circuit?
- How do we build a circuit to model a problem?
- What is Half/Full adder circuit?
- What are the benefits of simplification?
- What is Algebraic Simplification?
- What are Karnaugh Maps?

Notes

- Logic gate = electronic circuit in which the input can be either a single or multiple inputs. Similarly, we end up either single or multiple outputs
    - Outputs are logical function of the inputs
    - Output of a logic can be 0 (low output) or 1 (high output)

Definition of a gate

- A logic gate is defined as the basic element of circuits implementing a Boolean operation

- The most basic logic circuits are **OR** gates, **AND** gates and invertors, or **NOT** gates. All Boolean **functions** can be written in terms of these **three** logic operations.

AND Gate

- The **AND gate** produces a HIGH output (value 1) when all inputs are HIGH; otherwise, the output is LOW (value 0)
- For a 2-input gate, the AND gate is represented by the following electrical **notation** and **truth table**:

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

The AND operation is written as $f = x \cdot y$ or $f = xy$

OR Gate

- The **OR gate** produces a HIGH output (value 1) when any of the two inputs is HIGH; otherwise, the output is LOW (value 0)
- For a 2-input gate, the OR gate is represented by the following electrical **notation** and **truth table**:

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

The OR operation is written as $f = x + y$

Inverter Gate

- The **inverter gate**, also known as the **NOT gate**, produces an opposite output of the input. When the input is LOW (0), the output is HIGH (1). When the input is HIGH, the output is LOW
- The **inverter** gate is represented by the following electrical **notation** and **truth table**:

x	$\bar{x}$
0	1
0	1
1	0
1	1

The NOT operation is written as $f = \bar{x}$

Other gates

There are **four** additional gates which are the results of combinations of the basic gates:

XOR gate:
- true only when the values of the inputs differ

NAND gate:
- equivalent to "not AND"

NOR gate:
- equivalent to "not OR"

XNOR gate
- equivalent to a "not XOR".

Multiple input gates

AND, OR, XOR and XNOR operations are all **commutative** and **associative**
- They can be extended to more than two inputs

For example:
- the **XNOR gate** can be applied to 3 inputs as shown here

NAND and NOR operations are both commutative but not associative. Extending the number of inputs is less obvious in these cases.

In writing cascaded NAND and NOR operations, we must use the **correct** parentheses.

Representing De Morgan's laws

Theorem 1: $\overline{x \cdot y} = \bar{x} + \bar{y}$

Theorem 2: $\overline{x + y} = \bar{x} \cdot \bar{y}$

Definition of a circuit

- Combinational circuits or logic networks are a combination of logic gates designed to model Boolean functions.
- A combinational circuit is a circuit that implements a Boolean function.
- The logic values assigned to the output signals is a Boolean function of the current configuration of input signals.

Building a circuit from a function

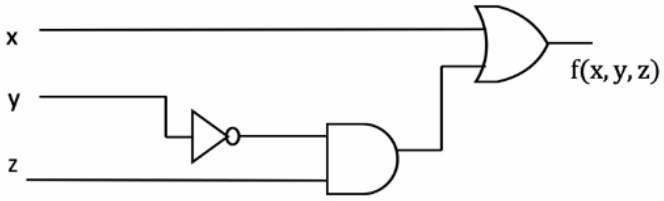
- Given a Boolean **function**, we can **implement** a logic **circuit** representing all the states of the function.
- Intuitively, we want to **minimise** the number of gates used in order to minimise the **cost** of the circuit.
- A Boolean function can be implemented **in different ways using** circuits.

Building a circuit from a function

- Let's consider the Boolean function f defined as:

$$f(x, y, z) = x + y'z$$

- f can be represented by the following circuit:



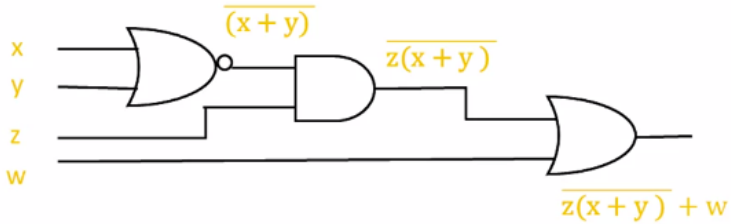
Writing a Boolean expression from a circuit

Given a logic network, we can work out its corresponding Boolean function as follows:

1. **label** all gate outputs that are a function of the input variables
2. **express** the **Boolean functions** for each gate in the first level
3. **repeat** the process until all the outputs of the circuit are written as Boolean expressions.

Example

Let's consider the following circuit:



Building a circuit to model a problem

Combinational circuits are useful for **designing systems** to solve specific problems, such as addition, multiplication, decoders and multiplexers.

The steps for building a circuit that solves a specific problem are:

1. **labelling** the inputs and outputs using variables
2. **modelling** the problem as a Boolean expression
3. **replacing** each operation by the equivalent logic gate.

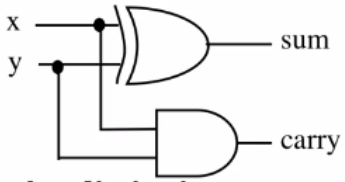
Building an adder circuit

Let's consider building an **adder** for two one-digit binary bits x and y.
From the truth table of this Boolean function, we know that:

- $\text{sum} = xy' + x'y = x \oplus y$
- $\text{carry} = xy$

x	y	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Which can be designed as a **half adder**:



This half adder has **limitations**:

- there is no provision for carry input
- this circuit is not useful for multi-bit additions.

Building a full adder circuit

To overcome its limitations, we can transform our half adder into to a **full adder** by including gates for processing the carry bit.

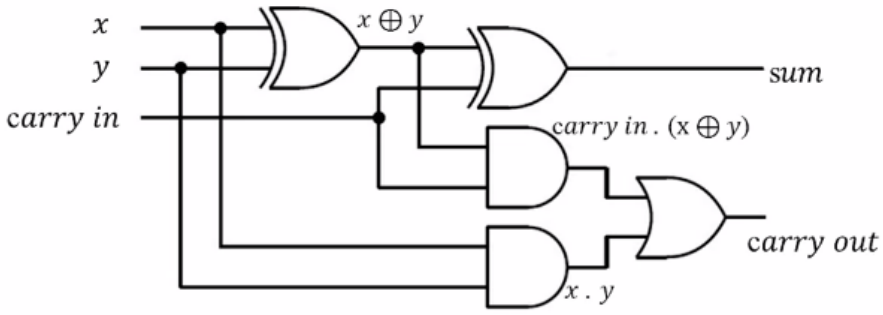
From the truth table for a full adder, we have:

$$\text{sum} = x \oplus y \oplus carry\ in$$
$$\text{carry out} = xy + carry\ in\ .\ (x \oplus y)$$

inputs			outputs	
x	y	carry in	sum	carry out
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

Building a full adder circuit

In order to **hide** the complexity of a circuit, we usually use a **box diagram** as a simple abstraction representing only inputs and outputs.



Benefits of simplification

Every function can be written as a **sum-of-product**, but this formulation is **not necessarily optimal** in terms of the number of gates and the depth of the circuit.

This is why circuits **need** to be **simplified**.

Simplification of circuits, also called **minimisation** or **optimisation**, is beneficial in circuit design, as it:

- **reduces** the global **cost** of circuits, by reducing the number of logic gates used
- might **reduce** the **time computation** cost of circuits
- allows **more** circuits to be fitted on the same chip.

Algebraic simplification

Algebraic simplification is based on the use of Boolean algebra theorems to represent and simplify the behaviour of Boolean functions.

To produce a sum-of-product expression, we usually need to use one or all of the following theorems:

- De Morgan's laws and involution
- distributive laws
- commutative, idempotent and complement laws
- absorption law.

Example

Let's consider the following Boolean expression

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

Using **De Morgan's laws** and **involution**:

$$E = ((xy)'' + z')((x' + z)' + (y' + z')')$$
$$= (xy + z')((x'' \cdot z') + y'' \cdot z'')$$
$$= (xy + z')(xz' + yz)$$

Using **distributive laws**:

$$E = xyxz' + xyyz + z'xz' + z'yz$$

Using **commutative**, **idempotent** and **complement** laws:

$$E = xyz' + xyz + xz' + 0$$

Using **absorption** law:

$$E = xyz' + xyz + xz' + 0$$

Example

Let's consider again the **full adder** circuit.

As seen previously, from the truth table we can build a sum-of-products form for the 2 functions:

$$\textbf{sum} = x'y'\,carry\ in + x'y\,carry\ in' + xy'\,carry\ in' + xy\,carry\ in$$
$$= carry\ in'\,(xy' + x'y) + carry\ in\,(xy + x'y')$$
$$= carry\ in'\,(x \oplus y) + carry\ in\,(x \oplus y)'$$
$$= carry\ in' \oplus x \oplus y$$

$$\textbf{\textit{carry out}} = x'y\,carry\ in + xy'\,carry\ in + xy\,carry\ in' + xy\,carry\ in$$
$$= (carry\ in + carry\ in')\,xy + carry\ in\,(xy' + x'y) = xy + carry\ in\,(x \oplus y)$$

inputs			outputs	
x	y	carry in	sum	carry out
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

Karnaugh maps

A Karnaugh map (or K-Map) is a **graphical** representation of Boolean functions and is different from a truth table. It can be used for expressions with 2, 3, 4 or 5 variables.

A K-Map is shown in an **array of cells** and cells differing by only one variable are adjacent.

The number of cells in a K-Map is the total number of possible input variable combinations, which equals 2^k.

For example:

- K-Map on the left represents the truth table on the right.

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

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

Example

Let's consider the Boolean function described in the truth table shown here:

- Since we have 3 variables, we need a 3-input K-Map, for which we identify all the 1's first
- Then, we group each 1 value with the maximum possible number of adjacent 1's to form a rectangle, power of 2 long (1, 2, 4, 8, ...)
- Then, we write a term for this rectangle.
- In this case, the minimised expression of **f** is: $x + yz$

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

	y'z'	y'z	yz	yz'
x'	0	0	1	0
x	1	1	1	1

Summary

In this week, we learned what gates are, the idea behind multiple gates together, what is a circuit is. Building on this, we looked at a Half/Full adder circuit, the steps on using a circuit to model a problem, the benefits of Circuit Simplification and Karnaugh Maps.