Boolean Algebra, Logic Gates, Circuits

CSIS 2226

What is Boolean Algebra?

- A minor generalization of propositional logic.
  - In general, an *algebra* is any mathematical structure satisfying certain standard algebraic axioms.
    - Such as associative/commutative/transitive laws, *etc.*
  - General theorems that are proved about an algebra then apply to *any* structure satisfying these axioms.
- *Boolean algebra* just generalizes the rules of propositional logic to sets other than {**T**,**F**}.
  - *E.g.*, to the set {0,1} of base-2 digits, or the set $\{V_L, V_H\}$ of low and high voltage levels in a circuit.
- We will see that this algebraic perspective lends itself to the design of *digital logic circuits*.

Claude Shannon's Master's thesis!

Boolean Algebra

- Sections of chapter 11:

  §1 – Boolean Functions

  §2 – Representing Boolean Functions

  §3 – Logic Gates

  §4 – Minimization of Circuits

§11.1 – Boolean Functions

- Boolean complement, sum, product.
- Boolean expressions and functions.
- Boolean algebra identities.
- Duality.
- Abstract definition of a Boolean algebra.

Complement, Sum, Product

- Correspond to logical NOT, OR, and AND.
- We will denote the two logic values as **0**:≡**F** and **1**:≡**T**, instead of **False** and **True**.
  - Using numbers encourages algebraic thinking.
- New, more algebraic-looking notation for the most common Boolean operators:

$\bar{x} :\equiv \neg x \qquad x \cdot y :\equiv x \wedge y \qquad x + y :\equiv x \vee y$

Precedence order→

Boolean Functions

- Let $B = \{\mathbf{0}, \mathbf{1}\}$, the set of Boolean values.
- For all $n \in \mathbf{Z}^+$, any function $f{:}B^n \rightarrow B$ is called a *Boolean function of degree n*.
- There are 2^{2^n} (wow!) distinct Boolean functions of degree n.
  - B/c ∃ 2^n rows in truth table, w. 0 or 1 in each.

Degree	How many	Degree	How many
0	2	4	65,536
1	4	5	4,294,967,296
2	16	6	18,446,744,073,709,551,616.
3	256		

Truth Tables

- A Boolean operator can be completely described using a truth table.
- The truth table for the Boolean operators AND and OR are shown at the right.
- The AND operator is also known as a Boolean product. The OR operator is the Boolean sum.

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

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

Truth Tables

- The truth table for the Boolean NOT operator is shown at the right.
- The NOT operation is most often designated by an overbar. It is sometimes indicated by a prime mark (‘) or an “elbow” (¬).

NOT X

X	$\overline{X}$
0	1
1	0

Boolean Functions

- The truth table for the Boolean function:

  $F(x, y, z) = x\overline{z}+y$

  is shown at the right.
- To make evaluation of the Boolean function easier, the truth table contains extra (shaded) columns to hold evaluations of subparts of the function.

$F(x,y,z) = x\overline{z}+y$

x	y	z	$\overline{z}$	$x\overline{z}$	$x\overline{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

Boolean Functions

- As with common arithmetic, Boolean operations have rules of precedence.
- The NOT operator has highest priority, followed by AND and then OR.
- This is how we chose the (shaded) function subparts in our table.

$F(x,y,z) = x\overline{z}+y$

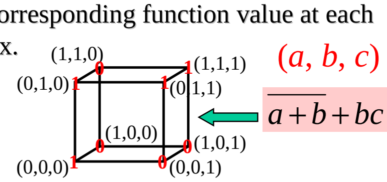
x	y	z	$\overline{z}$	$x\overline{z}$	$x\overline{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

Boolean Expressions

- Let $x_1, \ldots, x_n$ be n different Boolean variables.
  - n may be as large as desired.
- A *Boolean expression* (recursive definition) is a string of one of the following forms:
  - Base cases: **0**, **1**, x_1, …, or x_n.
  - Recursive cases: $\overline{E_1}$, (E_1E_2), or (E_1+E_2), where E_1 and E_2 are Boolean expressions.
- A Boolean expression represents a Boolean function.
  - Furthermore, *every* Boolean function (of a given degree) can be represented by a Boolean expression.

Hypercube Representation

- A Boolean function of degree n can be represented by an n-cube (hypercube) with the corresponding function value at each vertex.



Boolean equivalents, operations on Boolean expressions

- Two Boolean expressions e_1 and e_2 that represent the exact *same* function *f* are called *equivalent*. We write $e_1 \Leftrightarrow e_2$, or just $e_1 = e_2$.
  - Implicitly, the two expressions have the same value for *all* values of the free variables appearing in e_1 and e_2.
- The operators $\overline{}$, +, and · can be extended from operating on expressions to operating on the functions that they represent, in the obvious way.

Boolean functions and digital circuits

- Digital computers contain circuits that implement Boolean functions.
- The simpler that we can make a Boolean function, the smaller the circuit that will result.
  - Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
- With this in mind, we always want to reduce our Boolean functions to their simplest form.
- There are a number of Boolean identities that help us to do this.

Some popular Boolean identities

- Double complement:
  $\overline{\overline{x}} = x$
- Idempotent laws:
  $x + x = x, \quad x \cdot x = x$
- Identity laws:
  $x + \mathbf{0} = x, \quad x \cdot \mathbf{1} = x$
- Domination laws:
  $x + \mathbf{1} = \mathbf{1}, \quad x \cdot \mathbf{0} = \mathbf{0}$
- Commutative laws:
  $x + y = y + x, \quad x \cdot y = y \cdot x$
- Associative laws:
  $x + (y + z) = (x + y) + z$
  $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
- Distributive laws:
  $x + y \cdot z = (x + y) \cdot (x + z)$ ← Not true in ordinary algebras.
  $x \cdot (y + z) = x \cdot y + x \cdot z$
- De Morgan's laws:
  $\overline{(x \cdot y)} = \overline{x} + \overline{y}, \quad \overline{(x + y)} = \overline{x} \cdot \overline{y}$
- Absorption laws:
  $x + x \cdot y = x, \quad x \cdot (x + y) = x$

also, the Unit Property: $x + \overline{x} = \mathbf{1}$ and Zero Property: $x \cdot \overline{x} = \mathbf{0}$

Simplifying Boolean Functions

- We can use Boolean identities to simplify the function:
  as follows: $F(X,Y,Z) = (X + Y)(X + \overline{Y})\overline{(X\overline{Z})}$

$(X + Y)(X + \overline{Y})\overline{(X\overline{Z})}$	Idempotent Law (Rewriting)
$(X + Y)(X + \overline{Y})(\overline{X} + Z)$	DeMorgan's Law
$(XX + X\overline{Y} + XY + Y\overline{Y})(\overline{X} + Z)$	Distributive Law
$((X + Y\overline{Y}) + X(Y + \overline{Y}))(\overline{X} + Z)$	Commutative & Distributive Laws
$((X + 0) + X(1))(\overline{X} + Z)$	Inverse Law
$X(\overline{X} + Z)$	Idempotent Law
$X\overline{X} + XZ$	Distributive Law
$0 + XZ$	Inverse Law
XZ	Idempotent Law

Simplifying Boolean Functions

- Sometimes it is more economical to build a circuit using the complement of a function (and complementing its result) than it is to implement the function directly.
- DeMorgan's law provides an easy way of finding the complement of a Boolean function.
- Recall DeMorgan's law states:

$$\overline{(xy)} = \overline{x} + \overline{y} \quad \text{and} \quad \overline{(x+y)} = \overline{x}\,\overline{y}$$

Simplifying Boolean Functions

- DeMorgan's law can be extended to any number of variables.
- Replace each variable by its complement and change all ANDs to ORs and all ORs to ANDs.
- Thus, we find the the complement of:

$$F(X,Y,Z) = (XY) + (\overline{X}Z) + (Y\overline{Z})$$

is:

$$\overline{F}(X,Y,Z) = \overline{(XY) + (\overline{X}Z) + (Y\overline{Z})}$$
$$= \overline{(XY)}\,\overline{(\overline{X}Z)}\,\overline{(Y\overline{Z})}$$
$$= (\overline{X}+\overline{Y})(X+\overline{Z})(\overline{Y}+Z)$$

Duality

- The *dual* e^d of a Boolean expression e representing function f is obtained by exchanging $+$ with $\cdot$, and 0 with 1 in e.
  - The function represented by e^d is denoted f^d.
- **Duality principle:** If $e_1 \Leftrightarrow e_2$ then $e_1^d \Leftrightarrow e_2^d$.
  - **Example:** The equivalence $x(x+y) = x$ implies (and is implied by) $x + xy = x$.

Boolean Algebra, in the abstract

- A general *Boolean algebra* is *any* set B having elements **0**, **1**, two binary operators $\wedge$, $\vee$, and a unary operator $\neg$ that satisfies the following laws:
  - Identity laws: $x \vee \mathbf{0} = x$, $x \wedge \mathbf{1} = x$
  - Complement laws: $x \vee \neg x = \mathbf{1}$, $x \wedge \neg x = \mathbf{0}$
  - Associative laws: $(x \vee y) \vee z = x \vee (y \vee z)$, $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
  - Commutative laws: $x \vee y = y \vee x$, $x \wedge y = y \wedge x$
  - Distributive laws: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.

Note that B may generally have other elements besides **0**, **1**, and we have not fully defined any of the operators!

§11.2 – Representing Boolean Functions

- Sum-of-products Expansions
  - A.k.a. Disjunctive Normal Form (DNF)
- Product-of-sums Expansions
  - A.k.a. Conjunctive Normal Form (CNF)
- Functional Completeness
  - Minimal functionally complete sets of operators.

Sum-of-Products Expansions

- **Theorem:** Any Boolean function can be represented as a sum of products of variables and their complements.
  - **Proof:** By construction from the function's truth table. For each row that is 1, include a term in the sum that is a product representing the condition that the variables have the values given for that row.

Show an example on the board.

Literals, Minterms, DNF

- A *literal* is a Boolean variable or its complement.
- A *minterm* of Boolean variables $x_1,\ldots,x_n$ is a Boolean product of n literals $y_1\ldots y_n$, where y_i is either the literal x_i or its complement $\overline{x_i}$.
  - Note that at most one minterm can have the value 1.
- The *disjunctive normal form* (DNF) of a degree-n Boolean function f is the unique sum of minterms of the variables $x_1,\ldots,x_n$ that represents f.
  - A.k.a. the sum-of-products expansion of f.

Converting

- It is easy to convert a function to sum-of-products form using its truth table.
- We are interested in the values of the variables that make the function true (=1).
- Using the truth table, we list the values of the variables that result in a true function value.
- Each group of variables is then ORed together.

$F(x,y,z) = x\bar{z}+y$

x	y	z	$x\bar{z}+y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Converting

- The sum-of-products form for our function is:

$F(x,y,z) = \bar{x}y\bar{z}+\bar{x}yz+x\bar{y}\bar{z}+xy\bar{z}+xyz$

We note that this function is not in simplest terms. Our aim is only to rewrite our function in canonical sum-of-products form.

$F(x,y,z) = x\bar{z}+y$

x	y	z	$x\bar{z}+y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Conjunctive Normal Form

- A *maxterm* is a sum of literals.
- CNF is a *product-of-maxterms* representation.
- To find the CNF representation for f,
- take the DNF representation for complement $\neg f$,

$\neg f = \Sigma_i \Pi_j\, y_{i,j}$

- and then complement both sides & apply DeMorgan's laws to get:

$f = \Pi_i \Sigma_j\, \neg y_{i,j}$

Can also get CNF more directly, using the 0 rows of the truth table.

Functional Completeness

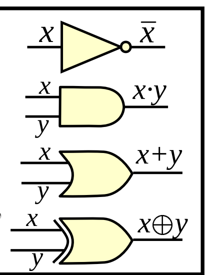
- Since every Boolean function can be expressed in terms of ·,+,‾, we say that the set of operators {·,+,‾} is *functionally complete*.
- There are smaller sets of operators that are also functionally complete.
  - We can eliminate either · or + using DeMorgan's law.
- NAND | and NOR ↓ are also functionally complete, each by itself (as a singleton set).
  - *E.g.*, $\neg x = x|x$, and $xy = (x|y)|(x|y)$.

§11.3 – Logic Gates

- Inverter, Or, And gate symbols.
- Multi-input gates.
- Logic circuits and examples.
- Adders, "half," "full," and *n*-bit.

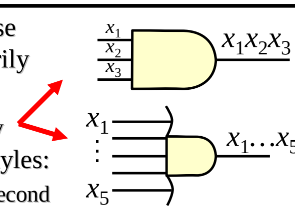
Logic Gate Symbols

- Inverter (logical NOT, Boolean complement).
- AND gate (Boolean product).
- OR gate (Boolean sum).
- XOR gate (exclusive-OR, sum mod 2).



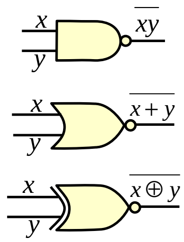
Multi-input AND, OR, XOR

- Can extend these gates to arbitrarily many inputs.
- Two commonly seen drawing styles:
  - Note that the second style keeps the gate icon relatively small.



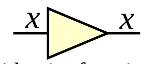
NAND, NOR, XNOR

- Just like the earlier icons, but with a small circle on the gate's output.
  - Denotes that output is complemented.
- The circles can also be placed on inputs.
  - Means, input is complemented before being used.



Buffer

- What about an inverter symbol *without* a circle?
- This is called a *buffer*. It is the identity function.
- It serves no logical purpose, but…
- It represents an explicit delay in the circuit.
  - This is sometimes useful for timing purposes.
- All gates, when physically implemented, incur a non-zero delay between when their inputs are seen and when their outputs are ready.



Combinational Logic Circuits

- **Note:** The correct word to use here is "combinat**ion**al," **NOT** "combinat**ori**al!"
  - Many sloppy authors get this wrong.
- These are circuits composed of Boolean gates whose outputs depend only on their most recent inputs, not on earlier inputs.
  - Thus these circuits have no useful memory.
    - Their state persists while the inputs are constant, but is irreversibly lost when the input signals change.

Combinational Circuit Examples

- Draw a few examples on the board:
  - Majority voting circuit.
  - XOR using OR / AND / NOT.
  - 3-input XOR using OR / AND / NOT.
- Also, show some binary adders:
  - Half adder using OR/AND/NOT.
  - Full adder from half-adders.
  - Ripple-carry adders.

§11.4 – Minimizing Circuits

- Karnaugh Maps
- *Don't care* conditions
- The Quine-McCluskey Method

Goals of Circuit Minimization

- (1) Minimize the number of primitive Boolean logic gates needed to implement the circuit.
  - Ultimately, this also roughly minimizes the number of transistors, the chip area, and the cost.
    - Also roughly minimizes the energy expenditure
      - among traditional irreversible circuits.
  - This will be our focus.
- (2) It is also often useful to minimize the number of combinational *stages* or logical *depth* of the circuit.
  - This roughly minimizes the *delay* or *latency* through the circuit, the time between input and output.

Minimizing DNF Expressions

- Using DNF (or CNF) guarantees there is always *some* circuit that implements any desired Boolean function.
  - However, it may be far larger than needed!
- We would like to find the *smallest* sum-of-products expression that yields a given function.
  - This will yield a fairly small circuit.
  - However, circuits of other forms (not CNF or DNF) might be even smaller for complex functions.