Boolean Logic & Circuits

Fall 2013

1. Introduction
   - Learning Outcomes for this Presentation
2. Boolean logic & Circuits, early work
3. Binary Number systems
   - 2's Complement representation of binary integers
   - Commonly used bases in computing
4. Summary

Learning Outcomes . . .

At the conclusion of this session, we will

- Explore the relationship between boolean operators and logical circuits.
- Show how hardware, logical gates, etc., compute boolean expressions.
- Demonstrate "normal" forms.
- Demonstrate the equivalence of circuits and boolean normal forms.
- Describe base-2 arithmetic, with an emphasis on addition and the construction of twos-complement (inverses) forms to perform subtraction.

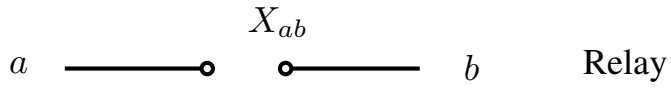
Computation in two-states

- Boolean expressions express two states: true or false, yes or no, on or off, open or closed, etc.
- Early in the twentieth century, Claude Shannon (and others) explored the equivalence of simple circuits and notions of computation.
- Our focus: circuits, gates, and then circuits.

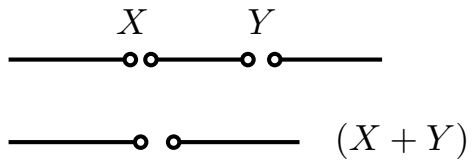
Do in class

Use Shannon's original diagrams to express the essential Boolean operators as simple circuits.

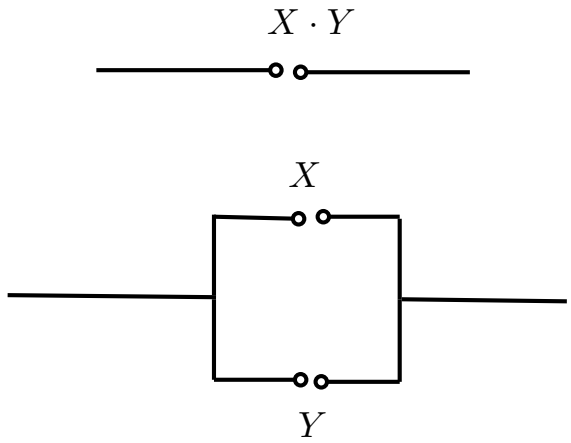
Equating circuits & Logic

- An "open circuit" is represented as a 1, impedance of 100%.
- A "closed circuit" is represented as a 0, impedance of 0%.
- Relay(s): 

- Series:



- Parallel:



Working postulates (draw these!)

1. A circuit is either open $X_{ab} = 1$, or closed $X_{ab} = 0$.
2. $0 \cdot 0 = 0$: A closed circuit *in parallel* with a closed circuit is a closed circuit.
3. $1 \cdot 1 = 1$: A *open* circuit *in parallel* with an *open* circuit is an *open* circuit.
4. $1 + 1 = 1$: An open circuit *in series* with an open circuit is an open circuit.
5. $0 + 0 = 0$: A closed circuit *in series* with a closed circuit is a closed circuit.
6. $0 \cdot 1 = 1 \cdot 0 = 0$: A closed circuit *in parallel* with an open circuit (in any order) is a closed circuit.
7. $1 + 0 = 0 + 1 = 1$: An open circuit *in series* with a closed circuit (in any order) is an open circuit;

Additional postulates . . .

Our system is incomplete with *negation*, which we will indicate with an accent, as in X' to indicate the negation of X.

- $X + X' = 1$
- $X \cdot X' = 0$
- $0' = 1$
- $1' = 0$
- $(X')' = X$.

Logical "duals . . ."

- For a given postulate, replacing the 0's with 1's and switching operators, e.g., from $\cdot$ to $+$ generates another postulate:

$$0 + 0 = 0 \Rightarrow 1 \cdot 1 = 1.$$

- These kinds of relationships are called "duals." Where else have we seen this?

Logical "duals . . ."

- For a given postulate, replacing the 0's with 1's and switching operators, e.g., from $\cdot$ to $+$ generates another postulate:

$$0 + 0 = 0 \Rightarrow 1 \cdot 1 = 1.$$

- These kinds of relationships are called "duals." Where else have we seen this?
- Think about De Morgan's Laws! Allow $+$ to be interpreted as a logical operator, say $\vee$, and $\cdot$ its "dual," $\wedge$, now

$$\neg(p \vee q) = \neg p \wedge \neg q \text{ and } \neg(p \wedge q) = \neg p \vee \neg q$$

Shannon's proof technique

- Shannon employed "universal induction" to prove many of his claims about circuits and logic.
- Universal induction, however, was "proof by exhaustion," in other words, he constructed *truth tables*.

Example

Let X and Y be two circuits and construct the truth table showing the possible states for both a series and a parallel arrangement:

Boolean Expressions as Circuits

- The take-away: any boolean expression can be represented as a circuit, and vice versa.
- Simplifications performed on boolean expressions translated into simplifications applied to circuits, saving components, reducing complexity, and enhancing efficiency and dependability.

A worked example

Example

Consider the expression: $p \vee \neg(q \wedge r)$:

First order of business: represent as a table, in *disjunctive normal form*:

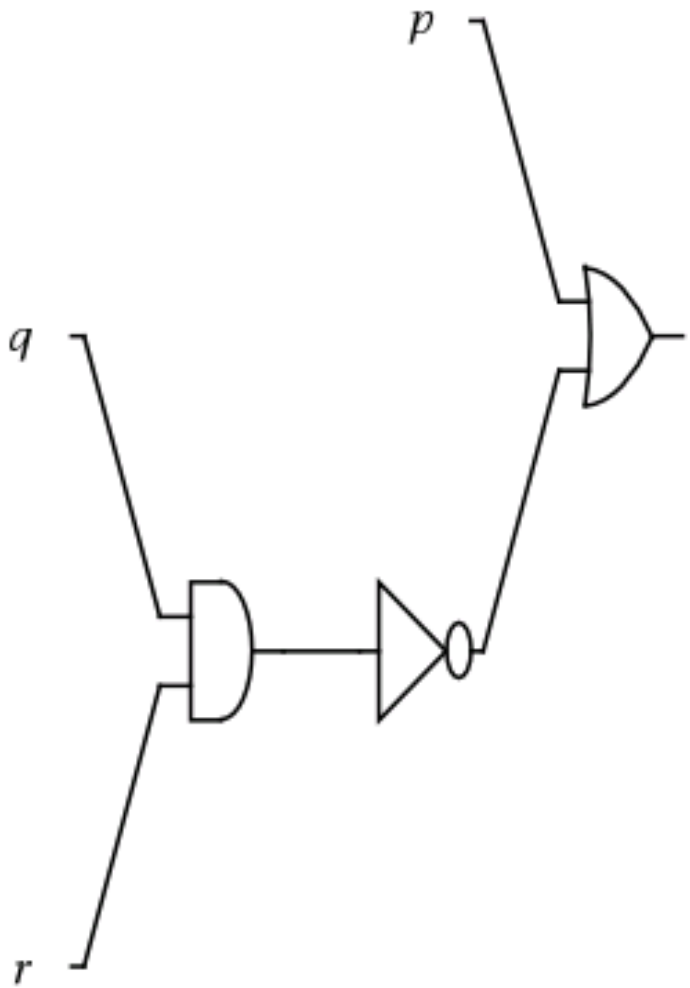
p	q	r	$p \vee \neg (q \wedge r)$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	T
F	F	T	T
F	F	F	T

Table of standard logical forms

Equivalent representations of $p \vee \neg(q \wedge r)$:

DNF	$p \vee (\neg q \wedge r)$
CNF	$(p \vee \neg q) \wedge (p \vee r)$
ANF	$(p \wedge r) \veebar (q \wedge r) \veebar (p \wedge q \wedge r) \veebar p \veebar r$
NOR	$(p \barwedge \neg q) \barwedge (p \barwedge r)$
NAND	$\neg p \barvee (\neg q \barvee r)$
AND	$\neg (\neg p \wedge q) \wedge \neg (\neg p \wedge \neg r)$
OR	$p \vee \neg (q \vee \neg r)$

Circuit for table



Circuits that "add" . . .

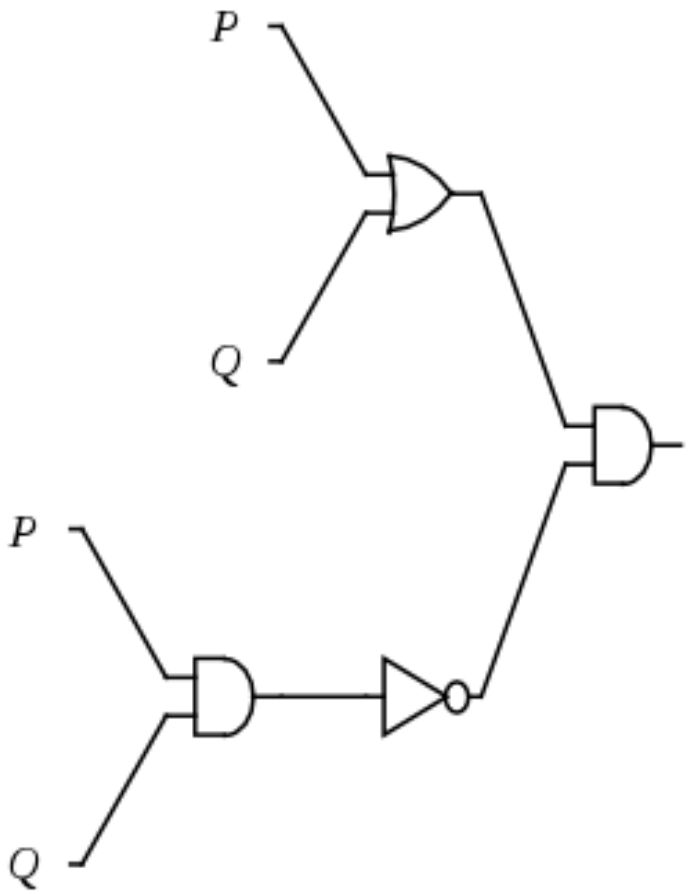
Consider the "half-adder," depicted in the table below:

P	Q	Carry	Sum
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Equivalences in tabular form

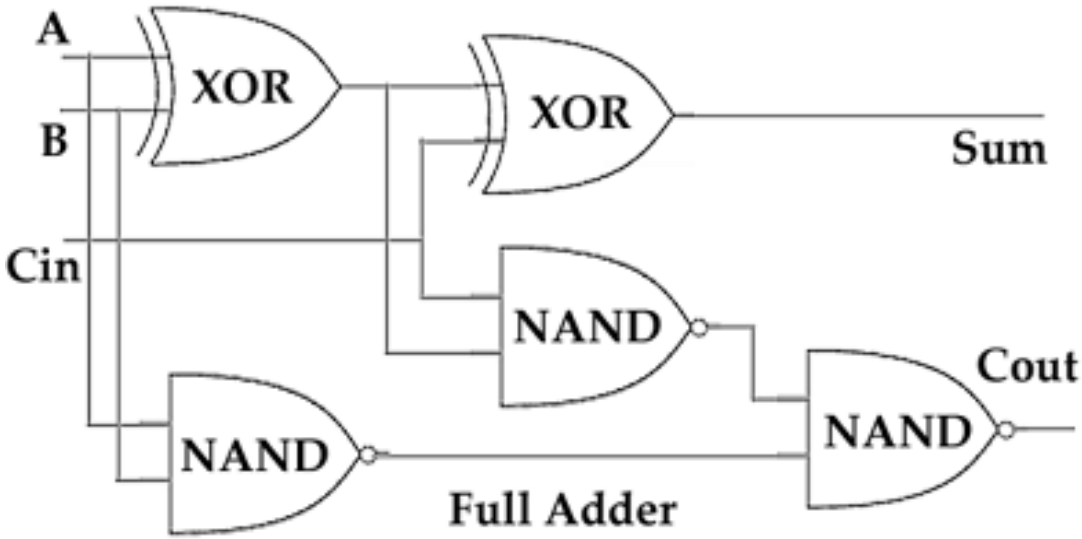
DNF	$(P \wedge \neg Q) \vee (\neg P \wedge Q)$
CNF	$(\neg P \vee \neg Q) \wedge (P \vee Q)$
ANF	$P \veebar Q$
NOR	$(\neg P \barwedge \neg Q) \barwedge (P \barwedge Q)$
NAND	$(P \barvee \neg Q) \barvee (\neg P \barvee Q)$
AND	$\neg (P \wedge Q) \wedge \neg (\neg P \wedge \neg Q)$
OR	$\neg (\neg P \vee Q) \vee \neg (P \vee \neg Q)$

Circuit diagram: half-adder



Adding more than one binary digit . . .

A *full-adder* is required to handle the carry bit!



Intermission

- Boolean logic is directly realized in physical circuits.
- Algebraic simplifications can be performed on expressions and directly applied to circuits.
- Mathematical characteristics, such as duality, symmetry/antisymmetry, equivalence, and complementarity find their way into circuit analysis and circuit design.

Encoding binary logic as numbers

- Let 0 represent a *closed* circuit (0 impedance), 1 an *open* circuit (100% impedance).
- A *string* of circuit *states* is represented as a sequence of 0's and 1's.
- Construct a *mapping* from these strings to numeric values

$\ldots$	1	0	1	0
2^n	2^3	2^2	2^1	2^0

which is interpreted as a base-10 digit number as follows:

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 10.$$

Basic observations about binary numbers

- The "largest" value that can be encoded in a binary string is 2^n, where n is a positive integer indicating the length of the string.
- Clearly, more binary digits are required to represent equivalent values in base-10 arithmetic.
- As presently defined, binary strings encode only *magnitude* (value, size).
- For simplicity's sake, we shall refer to binary strings as binary numbers from this point onward.

Essential arithmetic operations

- Binary numbers can be "added" in much the same way as base-10 numbers; recall how a full-adder circuit operates!
- Multiplying binary numbers by powers of 2 is easy—think about multiplying base-10 numbers by 10.
- Division by powers of 2 is likewise easy (but, what are the limitations?).
- Subtraction amounts to adding inverses! Ask what is an "inverse" and what is the "inverse" of a binary number?

Forming the additive inverse of a binary number

- Inverses are defined in relation to an operation: the additive inverse of the integer n is another integer, m, that when added to n gives the "identity" for addition: $n + m = 0$
- In binary arithmetic, the additive inverse of an integer is its "complement." And, conventionally, we use the 2's complement:

Definition

Given a positive integer a, the *2's complement* of a relative to a fixed-bit length n is the n-bit binary representation of $2^n - a$.

The usual algorithm modifies the original definition by explicitly adding 1 to the *1's complement*:

$$2^n - a = [(2^n - 1) - a] + 1$$

Forming the 2's complement

Construct the 2's complement of 57, which in binary is 00111001_2; assume 8 bit word length

- Subtracting 1 from 2^8 gives:

$$100000000_2 - 1_2 = 11111111_2.$$

Forming the 2's complement

Construct the 2's complement of 57, which in binary is 00111001_2; assume 8 bit word length

- Subtracting 1 from 2^8 gives:

$$100000000_2 - 1_2 = 11111111_2.$$

- Subtracting 57 from the 1's complement just "flips" its bits:

$$11111111_2 - 00111001_2 = 11000110_2.$$

Forming the 2's complement

Construct the 2's complement of 57, which in binary is 00111001_2; assume 8 bit word length

- Subtracting 1 from 2^8 gives:

$$100000000_2 - 1_2 = 11111111_2.$$

- Subtracting 57 from the 1's complement just "flips" its bits:

$$11111111_2 - 00111001_2 = 11000110_2.$$

- Finally, add 1 to form the 2's complement:

$$11000110_2 + 1_2 = 11000111_2.$$

Computing differences with 2's complements

Assuming that the 2's complement of 57 is equivalent to -57,solve $100 - 57$

- Translate 100 into an 8-bit binary number: $100_{10} = 01100100_2$.

Computing differences with 2's complements

Assuming that the 2's complement of 57 is equivalent to -57,solve $100 - 57$

- Translate 100 into an 8-bit binary number: $100_{10} = 01100100_2$.
- Subtracting 57 from 100 is the same adding its complement to 100:

$$01100100_2 + 11000111_2 = 00101011_2,$$

Computing differences with 2's complements

Assuming that the 2's complement of 57 is equivalent to -57,solve $100 - 57$

- Translate 100 into an 8-bit binary number: $100_{10} = 01100100_2$.
- Subtracting 57 from 100 is the same adding its complement to 100:

$$01100100_2 + 11000111_2 = 00101011_2,$$

- Because the leftmost digit is a 0, the result is positive, translate it directly to base 10.

Interpreting negative solutions

If we reverse the order of terms, we get a negative answer: $57 - 100$.

- Translate 57 as 00111001_2, and 100 as 01100100_2.

Interpreting negative solutions

If we reverse the order of terms, we get a negative answer: $57 - 100$.

- Translate 57 as 00111001_2, and 100 as 01100100_2.
- Construct the 2's complement of 100:

$$11111111_2 - 01100100_2 = 10011011 + 1 = 10011100_2$$

Interpreting negative solutions

If we reverse the order of terms, we get a negative answer: $57 - 100$.

- Translate 57 as 00111001_2, and 100 as 01100100_2.
- Construct the 2's complement of 100:

$$11111111_2 - 01100100_2 = 10011011 + 1 = 10011100_2$$

- Add -100 to 57:

$$10011100_2 + 00111001_2 = 11010101_2$$

Interpreting negative solutions

If we reverse the order of terms, we get a negative answer: $57 - 100$.

- Translate 57 as 00111001_2, and 100 as 01100100_2.
- Construct the 2's complement of 100:

$$11111111_2 - 01100100_2 = 10011011 + 1 = 10011100_2$$

- Add -100 to 57:

$$10011100_2 + 00111001_2 = 11010101_2$$

- Because the leftmost bit is a 1, use 2's complement arithmetic to translate this number, giving 00101011_2, which is 43 (but preceded with a negative sign because of the leftmost bit).

A mathematical property of the complement

The complement is an *involution*; recall Shannon's postulates, $(X')' = X$, which appears as $-(-n) = n$ in common number systems.

Definition

An *involution* is a function that when composed with itself gives the identity.

Do in class

Show that the definition of the 2's complement operation is an involution.

Some other commonly seen bases in computing

- Base-2 (binary) arithmetic is how computation happens at the "lowest level," but it takes a lot of real-estate to say a little.
- Other commonly encountered bases are multiples of 2, viz., base 8 (octal) and base 16 (hexadecimal).

Do in class

Construct some common base-2 numbers in base-8 and base-16, showing grouping patterns.

Summary

- Boolean logic formed the basis of computing machinery at the turn of the last century.
- Properties of the Boolean algebra permeate the study of computing.
- An understanding of the mathematical principles underlying the Boolean algebra deepens our understanding and appreciation for a variety of topic in contemporary computing.
- We should not be surprised to see these concepts again . . . perhaps in different contexts, throughout our study of computer science.