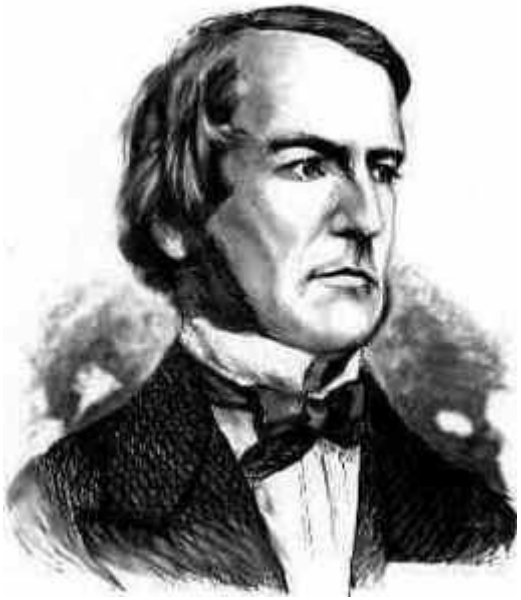
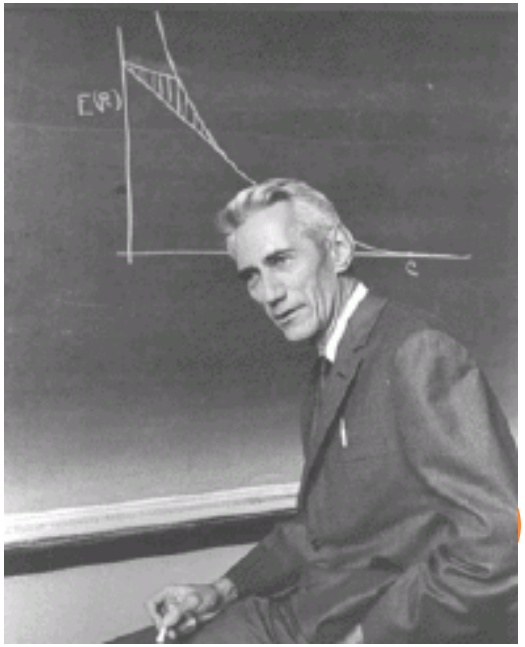
BOOLEAN ALGEBRA

Introduction

- 1854: Logical algebra was published by **George Boole** → known today as "Boolean Algebra"
  - It's a convenient way and systematic way of expressing and analyzing the operation of logic circuits.
- 1938: **Claude Shannon** was the first to apply Boole's work to the analysis and design of logic circuits.





Definition of Boolean Algebra

Boolean Algebra: A set of elements B with two operations. + (OR, $\cup$, $\vee$)
* (AND, $\cap$, $\wedge$),
satisfying the following 4 laws.

P1. Commutative Laws:
a+b = b+a; a*b = b*a,

P2. Distributive Laws:
a+(b*c) = (a+b)*(a+c); a*(b+c)= (a*b)+(a*c),

P3. Identity Elements: Set B has two distinct elements denoted as 0 and 1, such that a+0 = a; a*1 = a,

P4. Complement Laws:
a+a' = 1; a*a' = 0.

Boolean Operations & Expressions

- *Variable* – a symbol used to represent a logical quantity.
- *Complement* – the inverse of a variable and is indicated by a bar over the variable or the prime symbol.
- *Literal* – a variable, the complement of a variable and the constants (0,1).

BOOLEAN ALGEBRA VS SET OPERATIONS

- Identities
  - 0 = {}
  - 1 = Universe of the set
  - A+0 = A
  - A*1 = A

Operations

Multiplication ↔ Intersection

Addition ↔ Union

Complement (Negation) ↔ Complement

Example: U = {x | x is a positive integer}, A={x | x is odd}, B={x | x is even}, C={x | x is a prime number}

D = {x | x is even prime number} ↔ d = bc

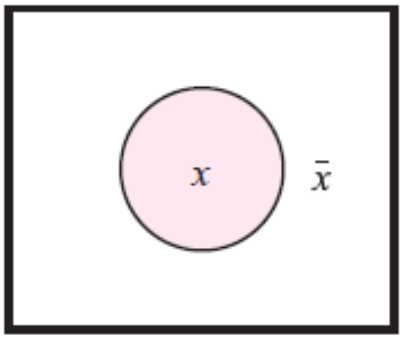
E = {x | x is odd or x is prime} ↔ e = a + c

$A \cup B = U \leftrightarrow a + b = 1$

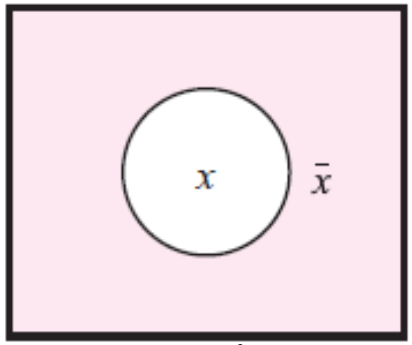
$A \cap B = \phi$ (empty set) ↔ ab = 0

$A^c = B \leftrightarrow a' = b$

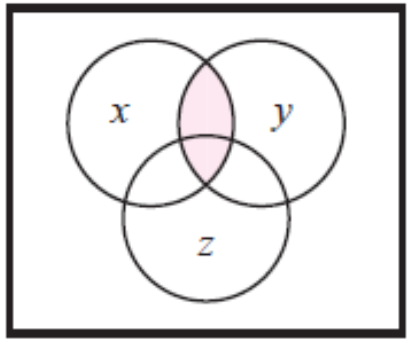
Venn Diagram Representation



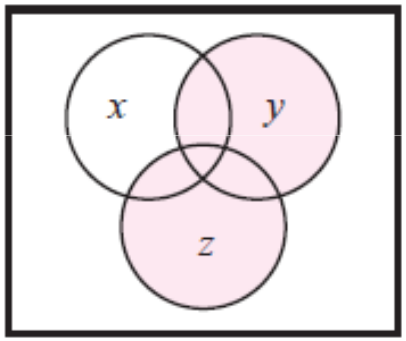
x



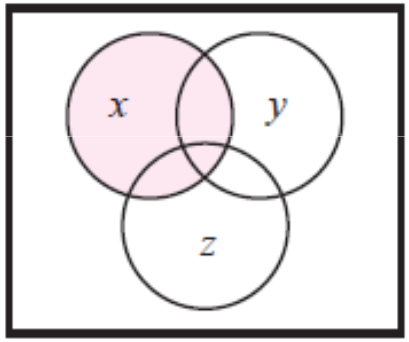
x'



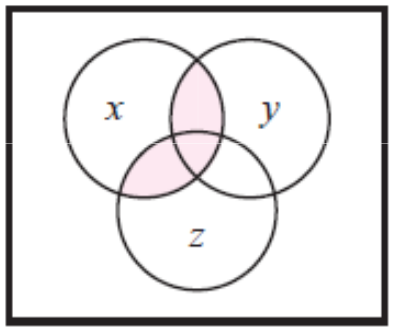
xy



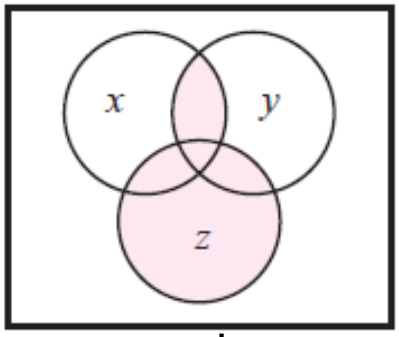
$y+z$



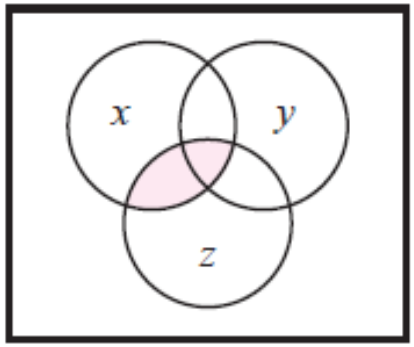
x



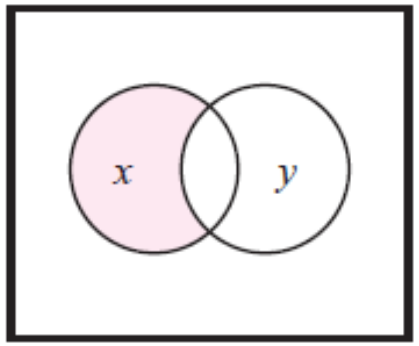
$x(y+z)$



$xy+z$



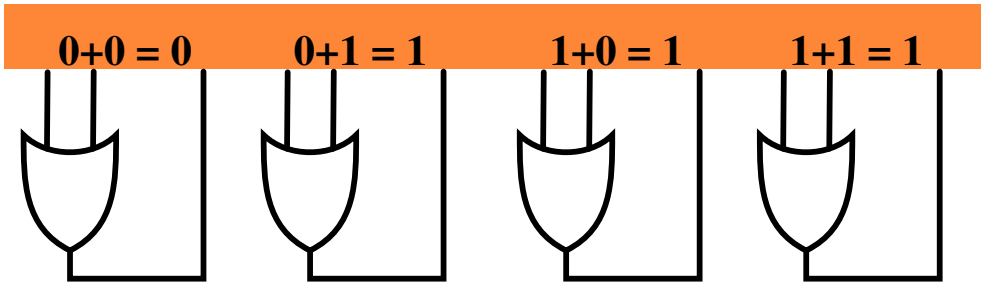
xz



xy'

Boolean Addition

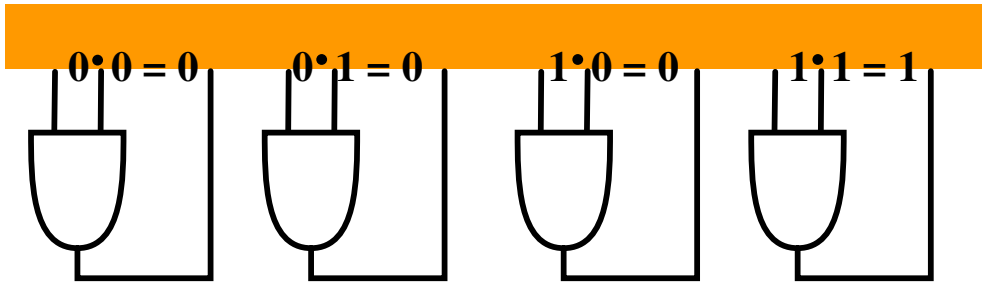
- Boolean addition is equivalent to the OR operation



- A *sum term* is produced by an OR operation with no AND ops involved.
  - i.e. $A + B, A + \overline{B}, A + B + \overline{C}, \overline{A} + B + C + \overline{D}$
  - A *sum term* is equal to 1 when one or more of the literals in the term are 1.
  - A *sum term* is equal to 0 only if each of the literals is 0.

Boolean Multiplication

- Boolean multiplication is equivalent to the AND operation



- A *product term* is produced by an AND operation with no OR ops involved.
  - i.e. $AB, A\overline{B}, AB\overline{C}, \overline{A}BC\overline{D}$
  - A *product term* is equal to 1 only if each of the literals in the term is 1.
  - A *product term* is equal to 0 when one or more of the literals are 0.

Truth Tables

Inputs		Outputs
x	*y*	*xy*
0	0	0
0	1	0
1	0	0
1	1	1

xy = x AND y = x * y
AND is true only if **both** inputs are true

Inputs		Outputs
x	*y*	*x* + *y*
0	0	0
0	1	1
1	0	1
1	1	1

x + y = x OR y
OR is true if **either** inputs are true

Inputs	Outputs
x	$\overline{x}$
0	1
1	0

x bar = NOT x
NOT inverts the bit
will denote x bar as x' (x prime)

NOR is NOT of OR

x	*y*	*x* NOR *y*
0	0	1
0	1	0
1	0	0
1	1	0

NAND is NOT of AND

x	*y*	*x* NAND *y*
0	0	1
0	1	1
1	0	1
1	1	0

XOR is true if both inputs **differ**

x	*y*	*x* XOR *y*
0	0	0
0	1	1
1	0	1
1	1	0

Boolean Expressions

- We form Boolean expressions out of Boolean operations on Boolean variables or Boolean values
  - So, like algebraic expressions, we can create more complex Boolean expressions as we might need
- Consider the expression:
  - F = x + y'z
- What is its truth table?

Notice that it is easier to derive the truth table for the entire expression by breaking it into subexpressions

So first we determine y'
next, y' z
finally, x + y'z

Inputs					Outputs
x	y	z	$\bar{y}$	$\bar{y}z$	$x + \bar{y}z = F$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

Basic Boolean Identities

- As with algebra, there will be Boolean operations that we will want to simplify
  - We apply the following Boolean identities to help
    - For instance, in algebra, x = y (z + 0) + (z * 0) can be simplified to just x = yz

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$(\overline{xy}) = \bar{x}+\bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

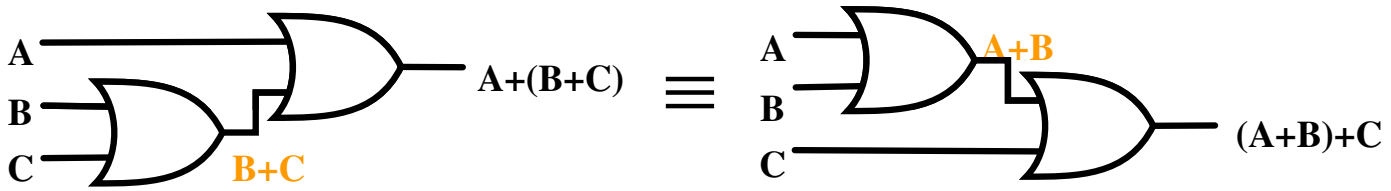
Commutative Laws

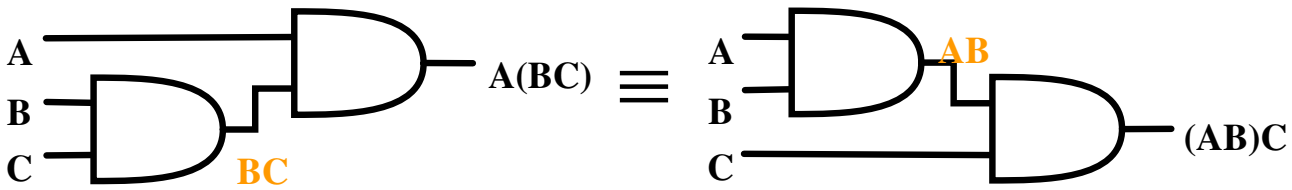
- The *commutative law of addition* for two variables is written as: $A+B = B+A$

- The *commutative law of multiplication* for two variables is written as: $AB = BA$

Associative Laws

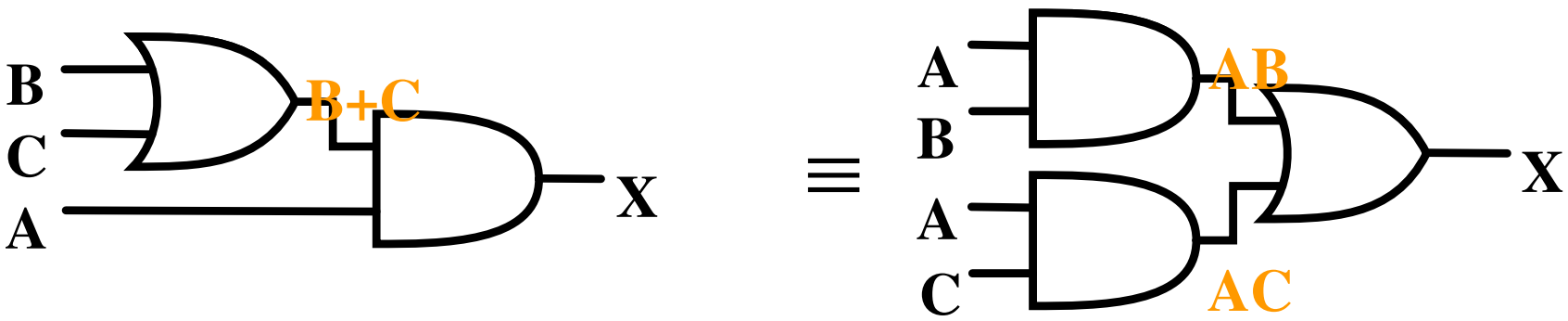
- The *associative law of addition* for 3 variables is written as: $A+(B+C) = (A+B)+C$
- The *associative law of multiplication* for 3 variables is written as: $A(BC) = (AB)C$





Distributive Laws

- The *distributive law* is written for 3 variables as follows: $A(B+C) = AB + AC$



X=A(B+C) **X=AB+AC**

DeMorgan's Theorems

- DeMorgan's theorems provide mathematical verification of:
  - the equivalency of the NAND and negative-OR gates
  - the equivalency of the NOR and negative-AND gates.
- The complement of two or more ANDed variables is equivalent to the OR of the complements of the individual variables.
- The complement of two or more ORed variables is equivalent to the AND of the complements of the individual variables.

NAND $\overline{X \bullet Y} = \overline{X} + \overline{Y}$ Negative-OR

NOR $\overline{X + Y} = \overline{X} \bullet \overline{Y}$ Negative-AND

In general, $\overline{\prod_k x_k} = \sum_k \overline{x_k}; \overline{\sum_k x_k} = \prod_k \overline{x_k}$

Some Examples

Example: use algebraic simplification rules to reduce x'yz+x'yz'+xz
x'yz + x'yz' + xz = x'y(z+z')+xz (distributive law) =
x'y(1)+xz (inverse law) = x'y+xz (identity law)

Example: xy+x'z+yz = xy+x'z+yz*1 (identity) = xy+x'z+yz*(x+x')
(inverse) = xy+x'z+xyz+x'yz (distributive) = xy(1+z)+x'z(y+1)
(distributive) = xy(1)+x'z(1) (null) = xy*1+x'z*1 (absorption)
= xy+x'z (identity)

Example: (x+y)(x'+y) = x'(x+y)+y(x+y) (distributive) = x'x+x'y+xy+yy (distributive)
= 0+x'y+xy+yy (inverse) = x'y+xy+yy (identity) = y(x'+x+y) (distributive)
= y(1+y) (inverse) = y(1) (identity) = y (idempotent)

Here we have an example specifically to see how DeMorgan's Law works

x	y	(xy)	$\overline{(xy)}$	$\overline{x}$	$\overline{y}$	$\overline{x}+\overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

DeMorgan's Law states that (xy) ' = x'+y'

Boolean expressions are equal if their truth tables give the same values – we see that here

Function Minimization using Boolean Algebra

- ***Examples***:

(a) $a + ab = a(1+b)=a$

(b) $a(a + b) = a.a + ab=a+ab=a(1+b)=a.$

(c) $a + a'b = (a + a')(a + b)=1(a + b) =a+b$

(d) $a(a' + b) = a.\ a' + ab=0+ab=ab$

(e) $ab + ab' = a(b+b') = a.1=a$

(f) $(a + b)(a + b') = a.a + ab' + ab+bb'$

$= a + ab' + ab + 0$

$= a + a(b' + b) + 0$

$= a + a1 + 0$

$= a + a = a$

(g) $(a + bc)' = a'(bc)' = a'(b' + c') = a'b' + a'c'$

DeMorgan's Theorems (Exercises)

- Apply DeMorgan's theorems to the expressions:

$$\overline{X \bullet Y \bullet Z}$$

$$\overline{(A + B + C)D}$$

$$\overline{X + Y + Z}$$

$$\overline{ABC + DEF}$$

$$\overline{\overline{X} + \overline{Y} + \overline{Z}}$$

$$\overline{A\overline{B} + \overline{C}D + EF}$$

$$\overline{\overline{W} \bullet \overline{X} \bullet \overline{Y} \bullet \overline{Z}}$$

$$\overline{\overline{A + B\overline{C}} + D(\overline{E + \overline{F}})}$$

DUALITY THEOREM (DE MORGAN'S DUALITY)

- **The Principle of Duality**

from Zvi Kohavi, Switching and Finite Automata Theory

"We observe that all the preceding properties are grouped in pairs. Within each pair one statement can be obtained from the other by interchanging the OR and AND operations and replacing the constants 0 and 1 by 1 and 0 respectively. Any two statements or theorems which have this property are called dual, and this quality of duality which characterizes switching algebra is known as the principle of duality. It stems from the symmetry of the postulates and definitions of switching algebra with respect to the two operations and the two constants. The implication of the concept of duality is that it is necessary to prove only one of each pair of statements, and its dual is henceforth proved."

- Recall the De Morgan's Theorems:

$(xy)' = x' + y' \leftrightarrow (x+y)'=x'y'$

- Consider basic Boolean identities in slide 11.

DUAL OF BOOLEAN EXPRESSIONS

(a) $f = (a + c')b + 0$

Dual of $f = (ac' + b)\cdot 1 = ac' + b$

(b) $g = xy+(w+z)'$

Dual of $g = (x + y)(wz)' = (x + y)(w' + z')$

(c) $h = ab + bc + ac$

Dual of $h = (a + b)(b + c)(a + c) = (b + ac)(a + c) = ab + bc + ac$ -> h is *self-dual.*

Application examples of Duality Theorem:

(a) $(x+y)(x'+y) = y \leftrightarrow xy + x'y = y$

(b) $(x+y)(x+y+z')+y' = (x+y)(1+z')+y' = x+y+y' = 1$

$\leftrightarrow (xy + xyz')y' = xyy' = 0$

(c) $abc + abc' = ab \leftrightarrow (a + b + c)(a + b + c') = a+b$

(d) $a'b + b'c + c'a = ab' + bc' + ca'$ (Prove it!)

$\leftrightarrow (a' +b)(b' +c)(c' +a) = (a+b')(b+c')(c+a')$

CONSENSUS THEOREM

- Consensus theorem states:

$XY + X'Z + YZ = XY + X'Z$

- The **YZ term is called the *consensus term and is*** redundant. The consensus term is formed from a PAIR OF TERMS in which a variable (X) and its complement (X') are present; the consensus term is formed by multiplying the two terms and leaving out the selected variable and its complement.
- The consensus of XY, $X'Z$ is YZ .
- Intuitively, if *YZ is true, then both Y and Z are true,* and therefore either *XY is true or X'Z is true.*

CONSENSUS THEOREM (CONT'D)

Proof

$XY + X'Z + YZ = XY + X'Z + (X + X')YZ$

$= XY + X'Z + XYZ + X'YZ$

$= (XY + XYZ) + (X'Z + X'YZ)$

$= XY(1 + Z) + X'Z(1 + Y)$

$= XY + X'Z$

- Exercise : Use the truth table to prove it.
- The consensus operator: ¢
- In general can write as:

$at_1 ¢ a't_2 = t_1t_2$

Dual of Consensus theorem

$(X+Y)\,(X'+Z)\,(Y+Z) = (X+Y)\,(X'+Z)$

The consensus of $(X+Y)(X'+Z)$ is $(Y+Z)$.

This can be derived using the duality theorem.

- Application Example: short cut for multiplication

$(X+Y)\,(X'+Z) = XZ + X'Y$

Only works when you have a variable (X) and its complement (X').

To PROVE this, lets do the distribution the long way.

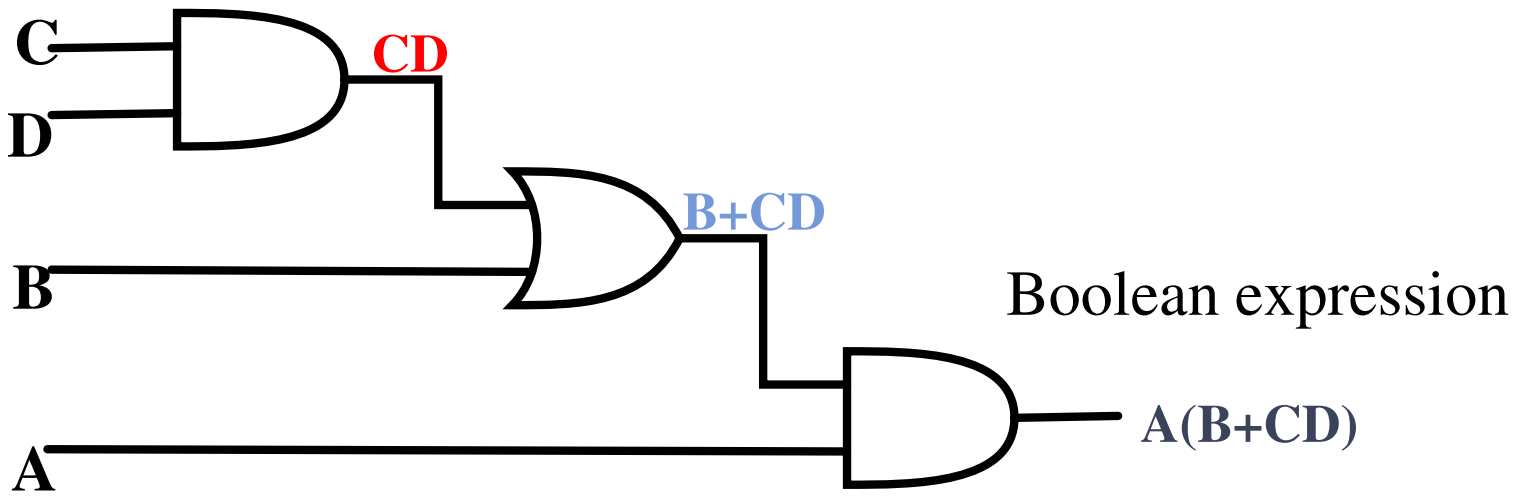
$(X+Y)(X'+Z) = X\,X' + XZ + X'Y + YZ = XZ + X'Y$

$X\,X'$ → 0

YZ → Redundant by consensus theorem

Boolean Analysis of Logic Circuits

- Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates so that the output can be determined for various combinations of input values.
- To derive the Boolean expression for a given logic circuit, begin at the left-most inputs and work toward the final output, writing the expression for each gate.



Constructing a Truth Table for a Logic Circuit

- Putting the results in truth table format

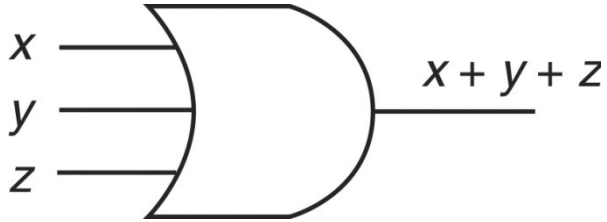
A(B+CD)=1

When A=1 and B=1 regardless of the values of C and D

When A=1 and C=1 and D=1 regardless of the value of B

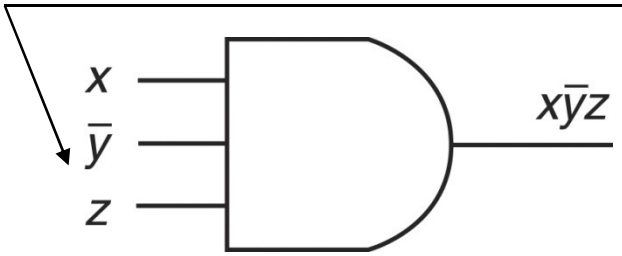
INPUTS				OUTPUT
A	B	C	D	A(B+CD)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

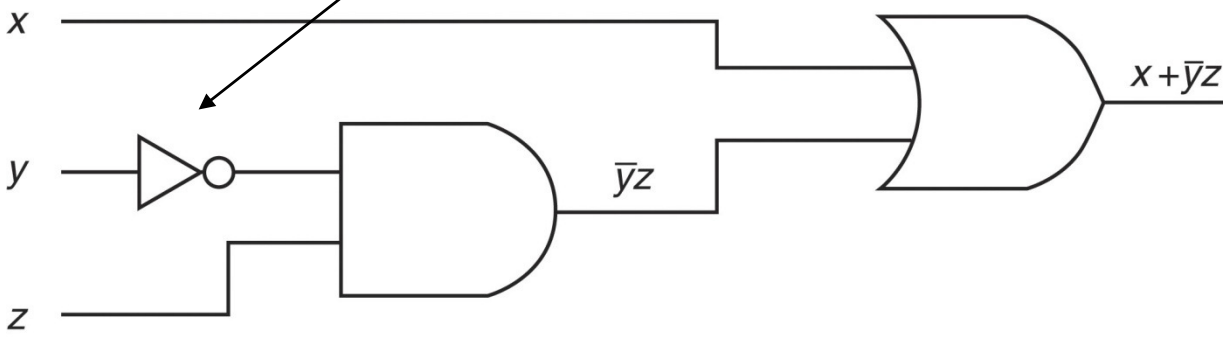
Some Circuits



AND and OR gates can have more than 2 inputs, as seen here

Notice for y', we can either use a NOT gate or simply specify that the value is y'



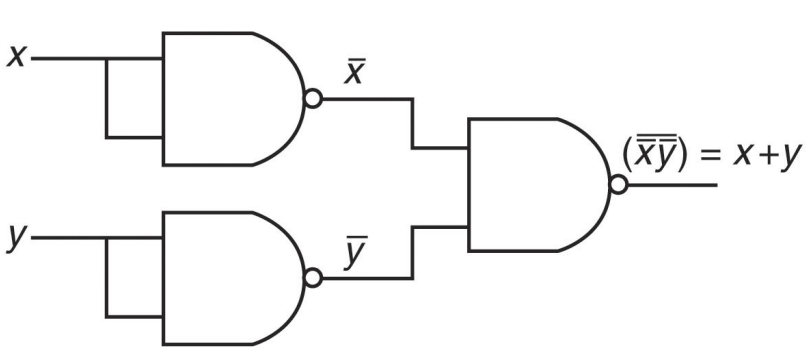


Here is the truth table for this last circuit

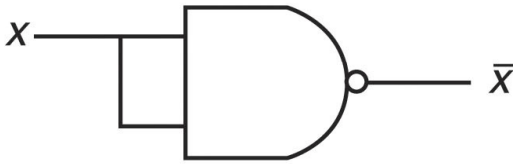
Inputs					Outputs
x	y	z	$\bar{y}$	$\bar{y}z$	$x + \bar{y}z = F$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

USING ONLY NAND

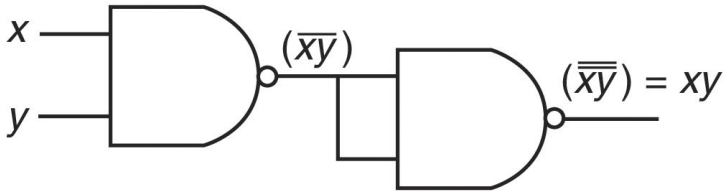
- NAND (and NOR) have unique properties different from the other Boolean operations
  - This allows us to use one or more NAND gates (or one or more NOR gates) and create gates that can compute AND, OR and NOT
    - See the examples below



OR Gate

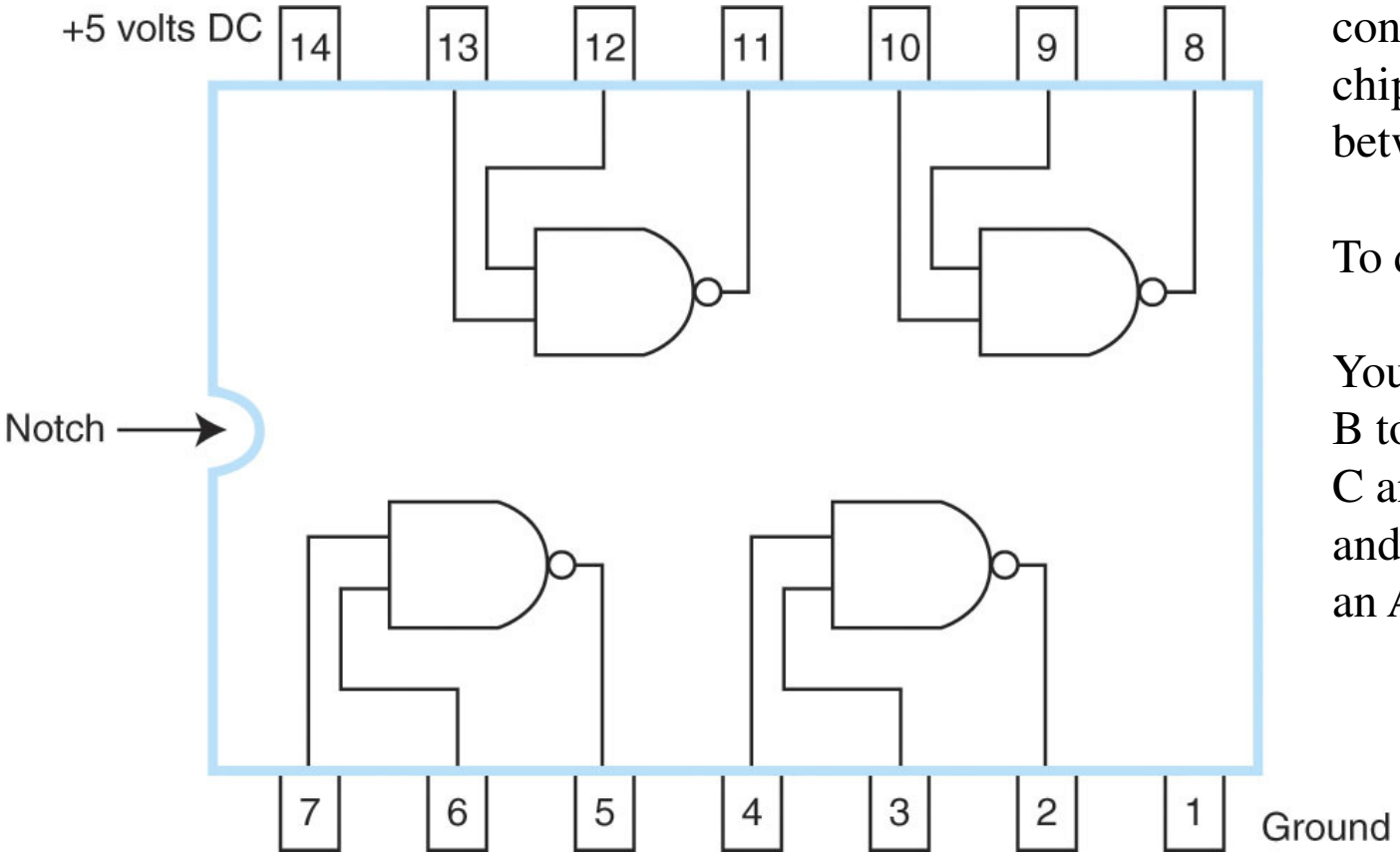


NOT Gate



AND Gate

NAND Chip



This is a NAND chip

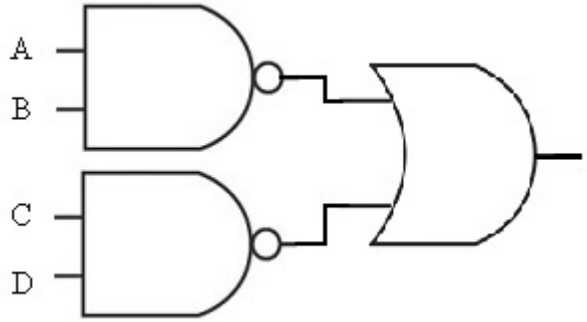
Early integrated circuits were several gates on a single chip, you would connect this chip to other chips by adding wires between the pins

To do (AB)'(CD)'

You would connect A and B to pins 7 and 6,
C and D to 4 and 3,
and send 5 and 2 to an AND chip

Combinational Circuits

- We will combine logic gates together for calculations
  - Example: (A*B)' and (C*D)' with an OR gate – shown to the right



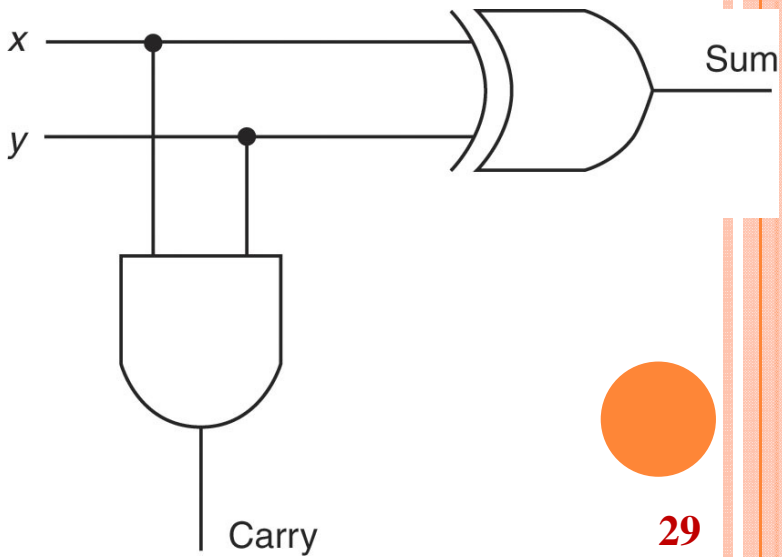
- The resulting circuit is a <u>combinational circuit</u>
  - Electrical current flows from one gate to the next
  - By combining gates, we can compute a Boolean expression
  - What we want to do is:
    - Derive the Boolean expression for some binary calculation (e.g., addition)
    - Then build the circuit using the various logic gates
  - This is how we will build the digital circuits that make up the ALU (arithmetic-logic unit) and other parts of the computer

An Example: Half Adder

- There are 4 possibilities when adding 2 bits together:
  - 0 + 0 0 + 1 1 + 0 1 + 1
    - In the first case, we have a sum of 0 and a carry of 0
    - In the second and third cases, we have a sum of 1 and a carry of 0
    - In the last case, we have a sum of 0 and a carry of 1
    - These patterns are demonstrated in the truth table above to the right
  - Notice that sum computes the same as XOR and carry computes the same as AND
  - We build an Adder using just one XOR and one AND gate

Inputs		Outputs	
x	*y*	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The truth table for Sum and Carry and a circuit to compute these



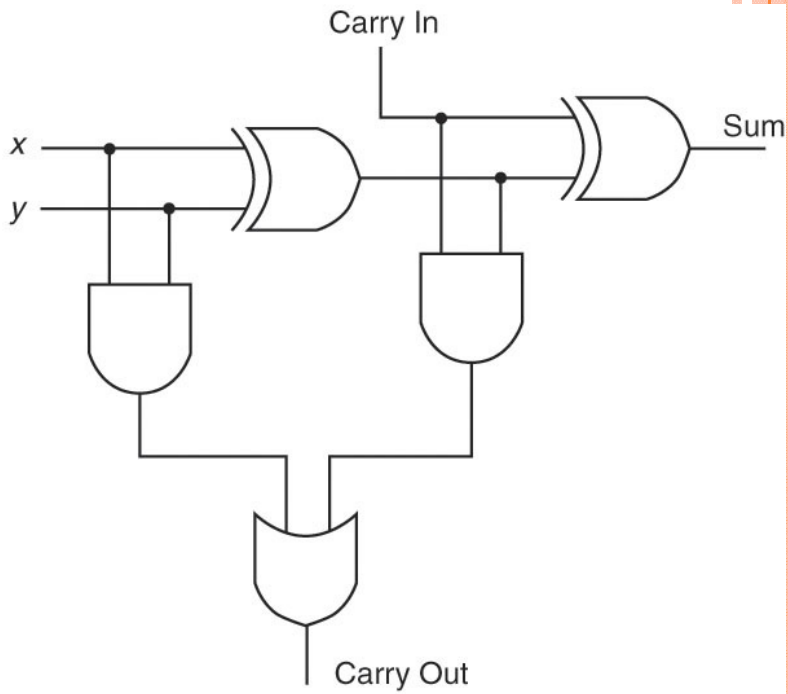
Full Adder

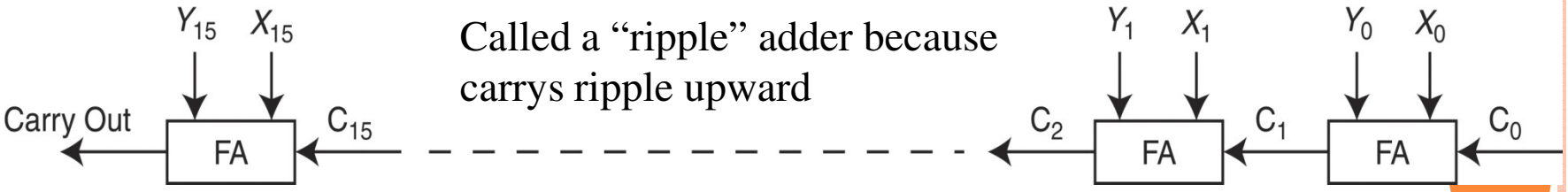
- The half adder really only does half the work
  - adds 2 bits, but only 2 bits
- If we want to add 2 n-bit numbers, we need to also include the carry in from the previous half adder
  - So, our circuit becomes more complicated
- In adding 3 bits (one bit from x, one bit from y, and the carry in from the previous addition), we have 8 possibilities
  - The sum will either be 0 or 1 and the carry out will either be 0 or 1
  - The truth table is given to the right

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

- The sum is 1 only if one of x, y and carry in are 1, or if all three are 1, the sum is 0 otherwise
- The carry out is 1 if two or three of x, y and carry in were 1, 0 otherwise
  - The circuit to the right captures this by using 2 XOR gates for Sum and 2 AND gates and an OR gate for Carry Out
- We combine several full adders together to build an Adder, as shown below:





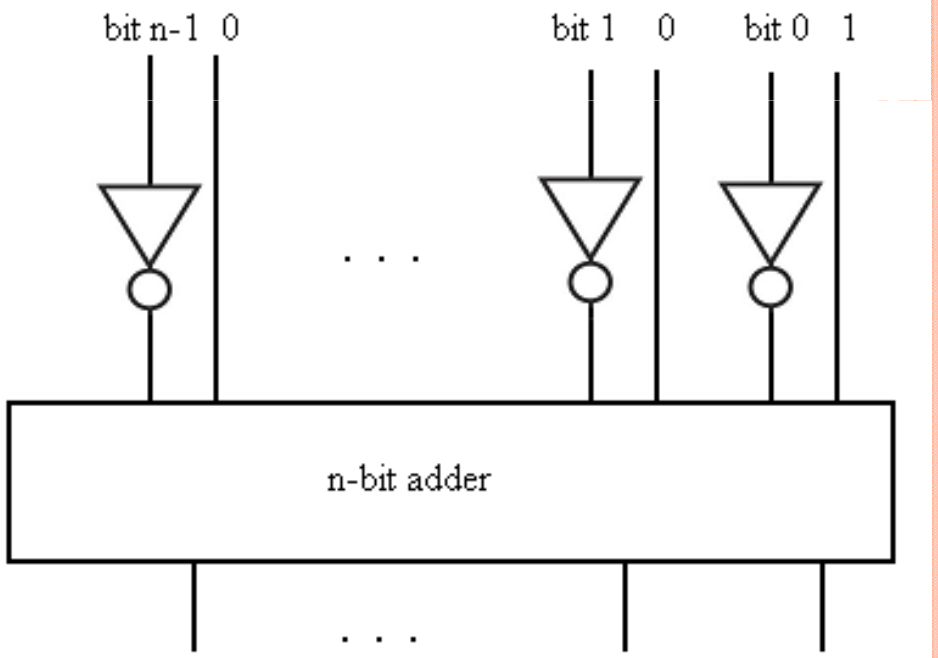
Called a "ripple" adder because carrys ripple upward

A 16-bit adder, comprised of 16 Full Adders connected so that each full adder's carry out becomes the next full adder's carry in

COMPLEMENTOR

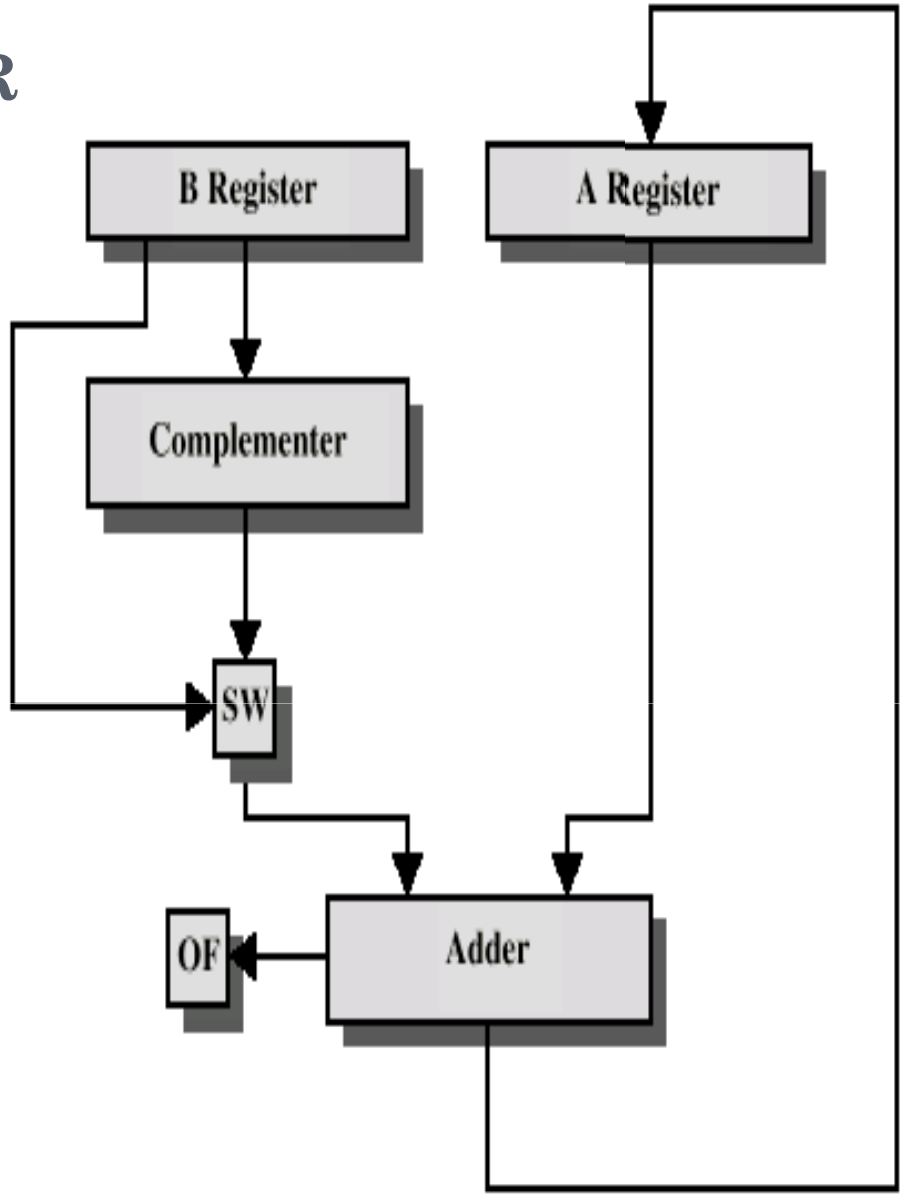
- Let's design another circuit to take a two's complement number and negate it (complement it)
  - Change a positive number to a negative number
  - Change a negative number to a positive number

- Recall to do this, you flip all of the bits and add 1
  - To flip the bits, we pass each bit through a NOT gate
  - To add one, send it to a full adder with the other number being 000…001



ADDER/SUBTRACTER

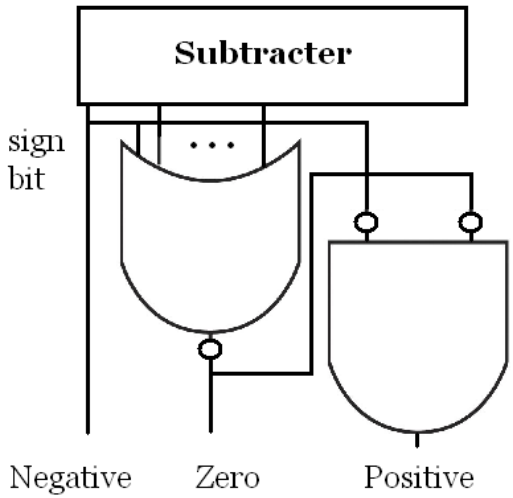
- Recall that
  - two's complement subtraction can be performed by negating the second number and adding it to the first
    - We revise our adder as shown to the right
      - It can now perform addition (as normal)
      - Or subtraction by sending the second number through the complementor



OF = overflow bit
SW = Switch (select addition or subtraction)

The switch (SW) is a multiplexer, covered in a few slides

COMPARATOR

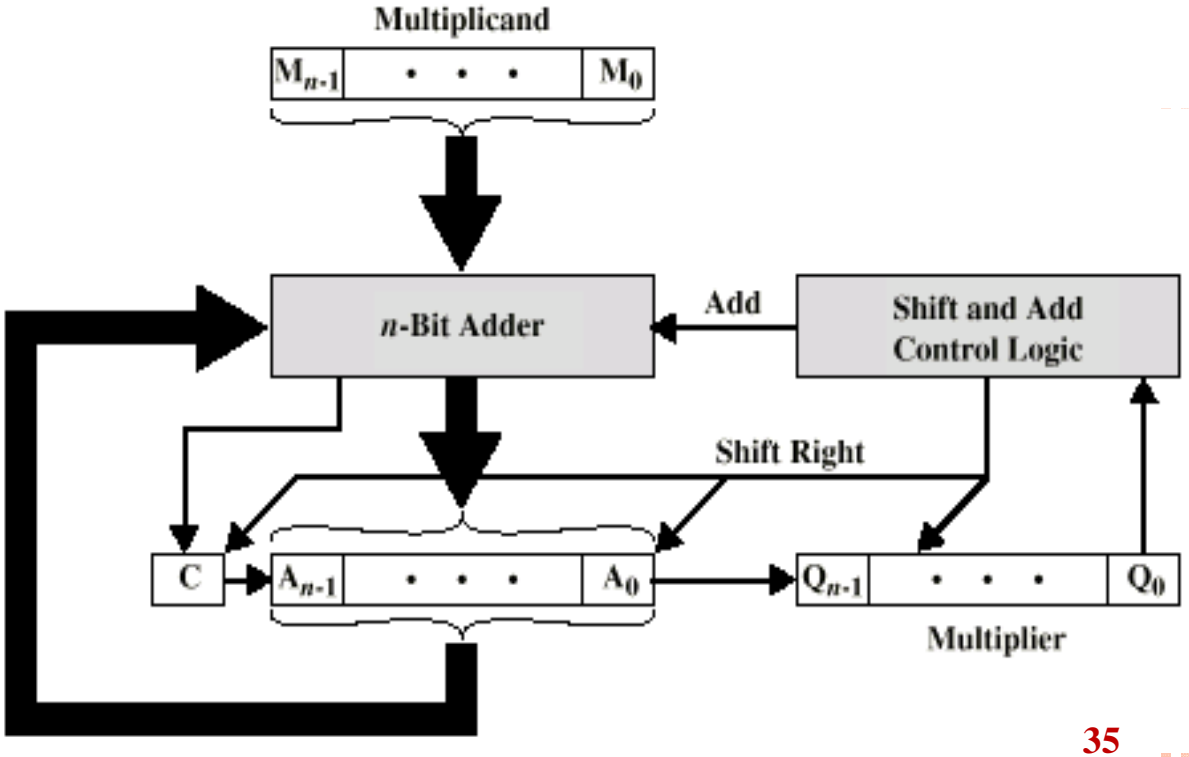


- have covered + and -, how about <, >, =
- To compare A to B, we use a simple tactic
  - Compute A – B and look at the result
    - if the result is -, then A < B
    - if the result is 0, then A = B
    - if the result is +, then A > B
    - if the result is not 0, then A != B
  - how do we determine if the result is -? look at the sign bit, if the sign bit is 1, then the result is negative and A < B
  - how do we determine if the result is 0? are all bits of the result 0? if so, then the result is 0 and A = B
    - we will build a zero tester which is simply going to NOR all of the bits together
  - how do we determine if the result is +? if the result of A – B is not negative and not 0, it must be positive, so we negate the results of the first two and pass them through an AND gate
- The comparator circuit is shown above (notice that the circuit outputs 3 values, only 1 of which will be a 1, the others must be 0)
  - NOTE: to compute !=, we can simply negate the zero output

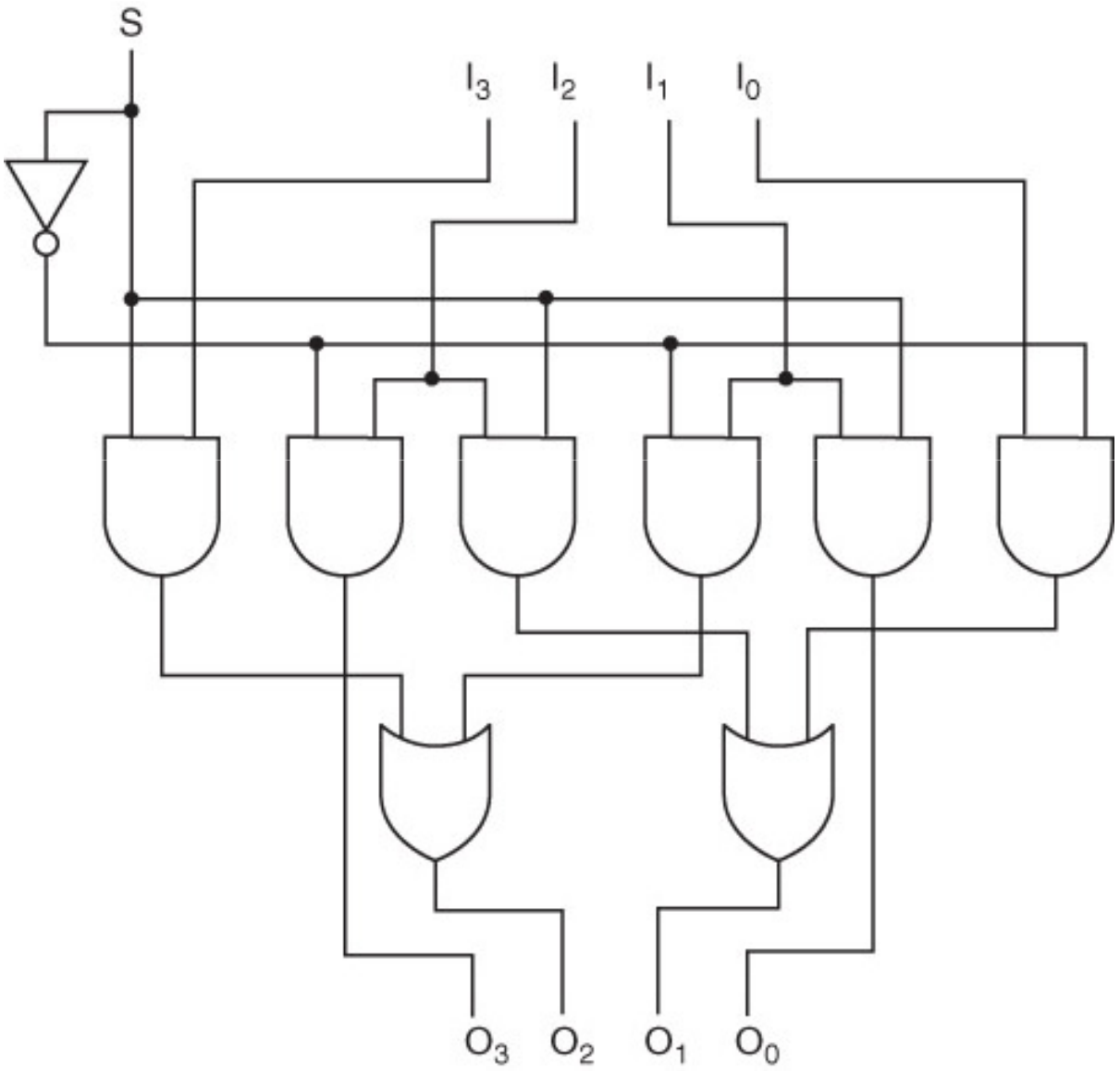
MULTIPLIER

- The circuit below is a multiplication circuit
  - Given two values, the multiplicand and the multiplier, both stored in temporary registers
  - The addition takes place by checking the Q0 bit and deciding whether to add the multiplicand to the register A or not, followed by right shifting the carry bit, A and Q together

- shift/add control logic
  - set counter = n
  - compare Q0 to 1
    - if equal, signal adder to add
  - signal the shifter to shift
  - decrement counter
  - repeat until counter = 0



4-BIT SHIFTER



Shift $I_3I_2I_1I_0$ to the left or the right to become $O_3O_2O_1O_0$ based on S

PARITY FUNCTIONS

x	y	z	Parity Bit
0	0	0	1
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	0

What will this circuit look like?

Notice that parity = Not(sum)

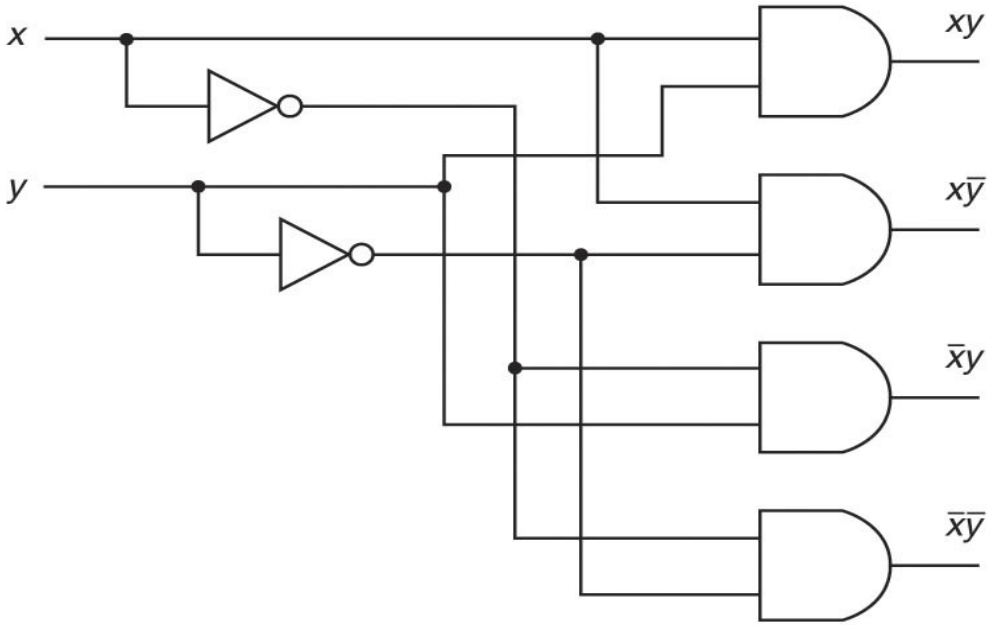
- The table on the left shows the Boolean values to indicate what parity value a 3 bit input should have
- The table on the right indicates whether an error should be signaled upon receiving 3 inputs and the parity info

The error detection circuit is more complex and will require that we either build it as a sum of products or use a simplification method

x	y	z	P	Error detected?
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

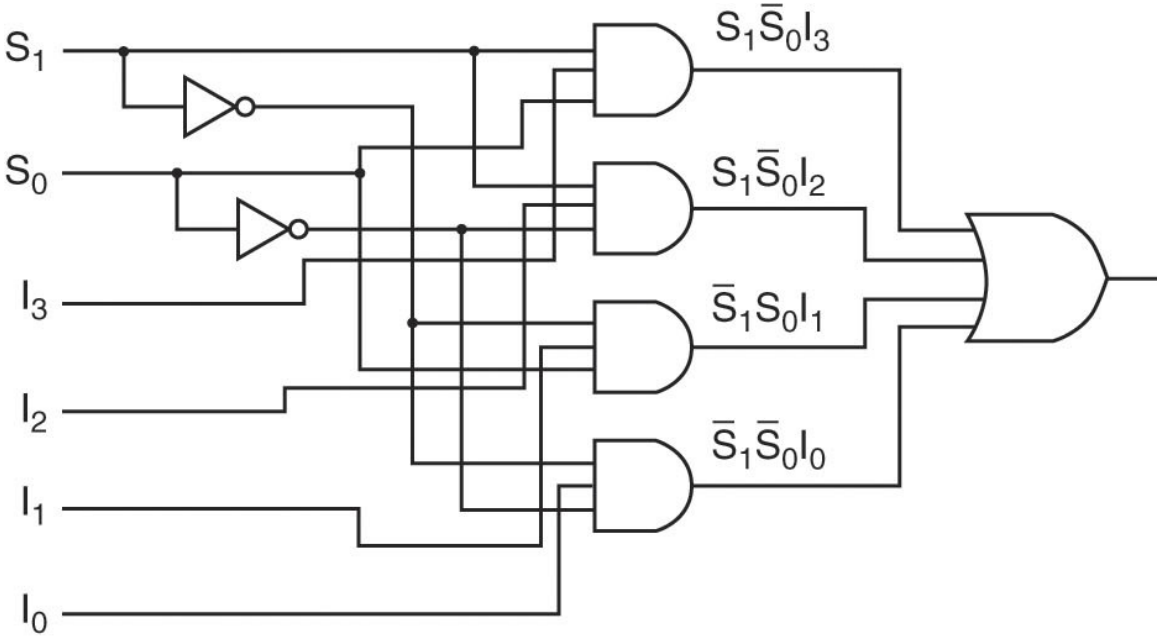
A Decoder

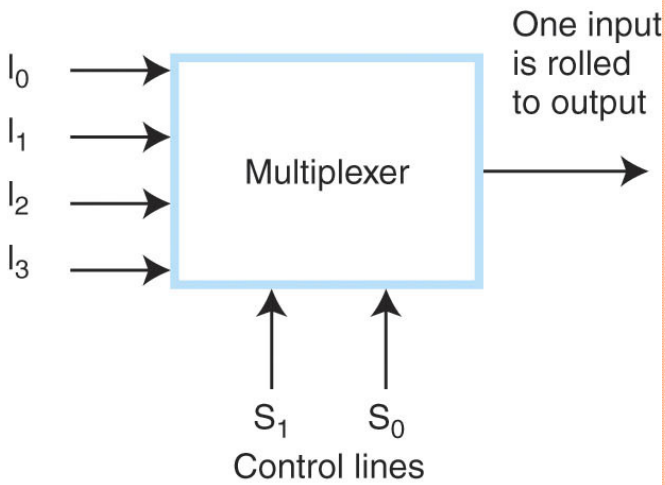
- The Decoder is a circuit that takes a binary pattern and translates it into a single output
  - This is often used to convert a binary value into a decimal value
    - For an n-bit input, there are 2^n outputs
    - Below is a 2 input – 4 output decoder
      - if input = 01, the second line (x'y) on the right has current
      - the line 01 would be considered line 1, where we start counting at 0



A Multiplexer

- Multiplexer (abbreviated as MUX) is used to select from a group of inputs which one to pass on as output
  - Here, 1 of 4 single-bit inputs is passed on using a 2-bit selector (00 for input 0, 01 for input 1,10 for input 2, 11 for input 3)
    - While this circuit is more complex than previous ones, this is simplified for a MUX – imagine what it would look like if we wanted to pass on 16 bits from 1 of 4 inputs





A related circuit is the demultiplexer (DEMUX) – it receives 1 input and a select and passes the input onto one of several outputs

A Simple 2-bit ALU

Putting all these ideas together

We have a 2-bit ALU

Given 2 values, A and B, each of which are 2 bits (A0, A1, B0, B1) and a selection from the control unit (f0, f1)

This circuit computes
A+B (if f0 f1 = 00)
NOT A (if f0 f1 = 01)
A OR B (if f0 f1 = 10)
A AND B (if f0 f1 = 11)

And passes the result out as C0 C1 along with overflow if the addition caused an overflow