# Logic Gates

- A gate is an electronic device that produces a result based on one or more digital input values.
  - In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
  - Integrated circuits contain collections of gates suited to a particular purpose.
- Digital computer circuits employ logic gates to implement Boolean functions.

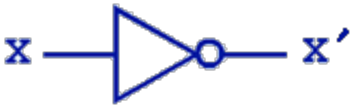
- The three simplest gates are the AND, OR, and NOT gates.


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 |




NOT X

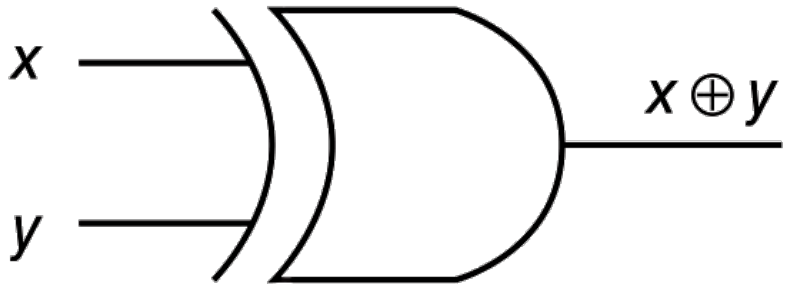
| X | X′ |
|---|---|
| 0 | 1 |
| 1 | 0 |

- They correspond directly to their respective Boolean operations, as shown in their truth tables.

- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.

**X XOR Y**

| X | Y | X ⊕ Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

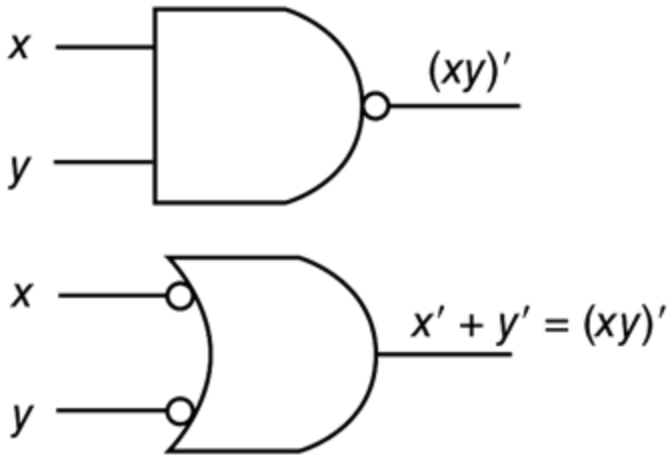



**The symbol ⊕ denotes the XOR operator.**

- Two other important gates are the NAND and NOR gates. The truth table for the NAND gate is shown below:

X NAND Y

| X | Y | X NAND Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

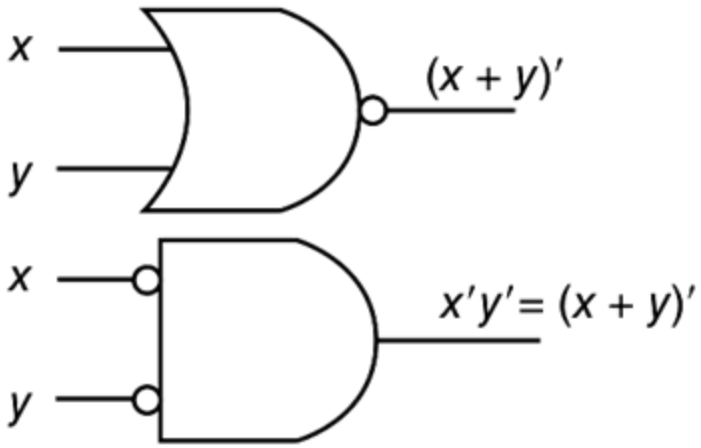



- The open circles are inversion bubbles. The two gates above are equivalent based on DeMorgan's theorem.

- The truth table for the NOR gate is shown below along with two equivalent implementations:

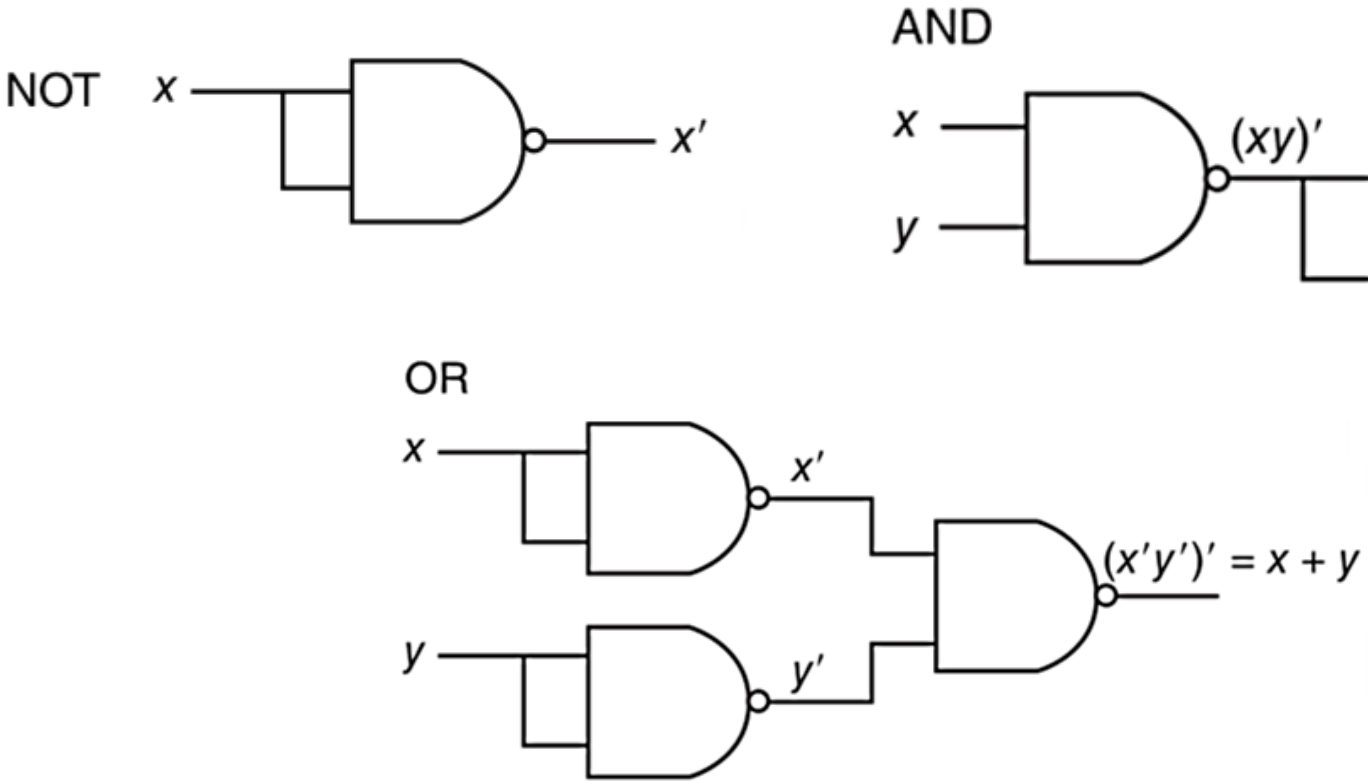
X NOR Y

| X | Y | X NOR Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

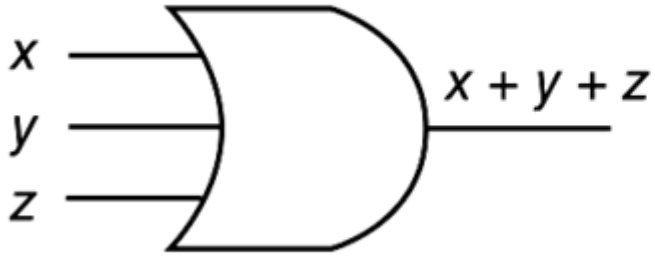


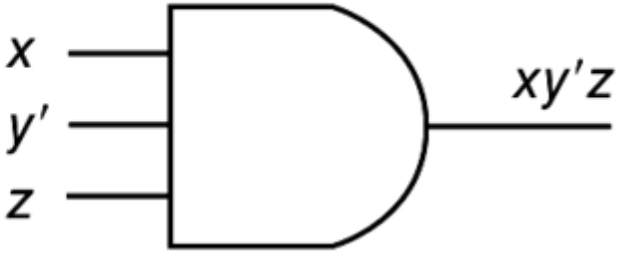

- NAND and NOR gates are said to be *universal* gates because they can be used to implement any logic function.

- The examples below show how NAND gates alone can be used to implement NOT, AND, and OR functions:

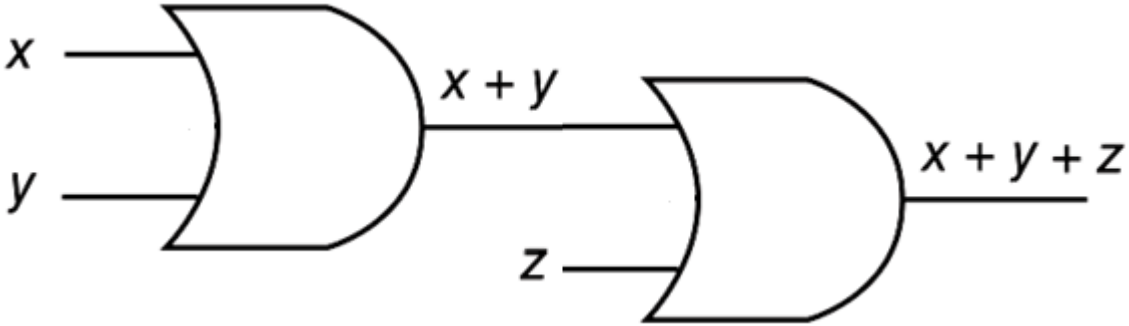


- Here are examples of 3-input gates:

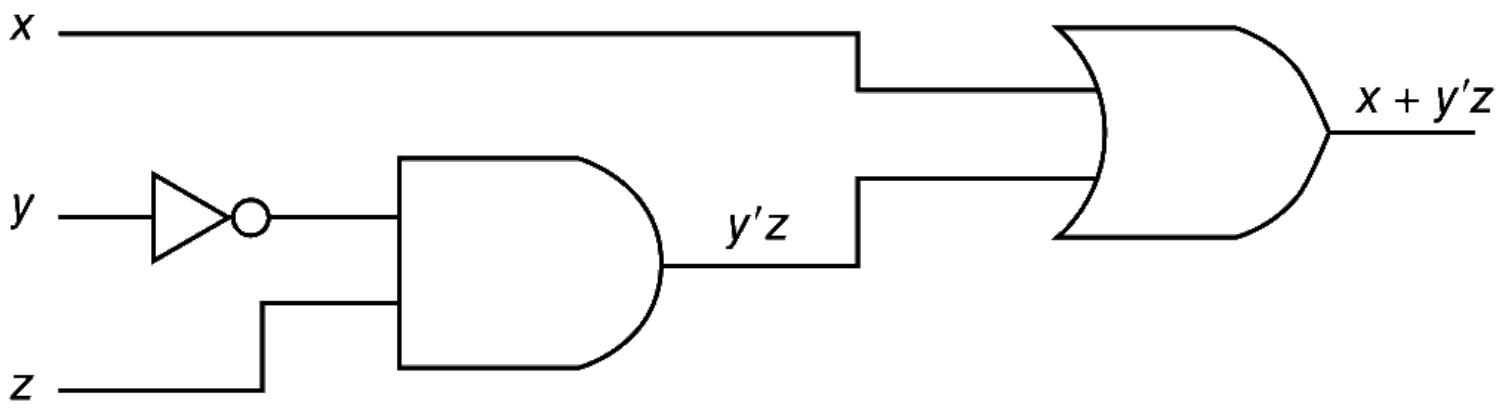



- However the same result can be generated using multiple 2-input gates:

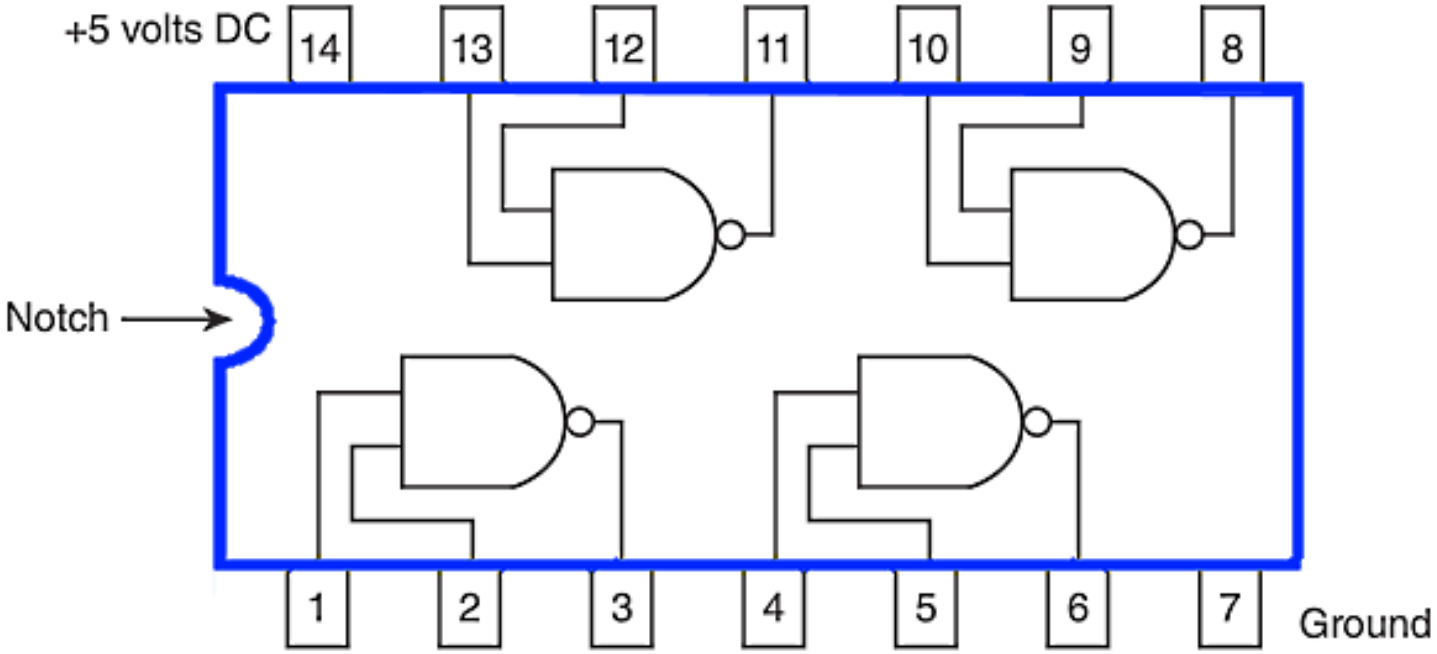


- The circuit below implements the Boolean function **F(x,y,z) = x + y′z**:



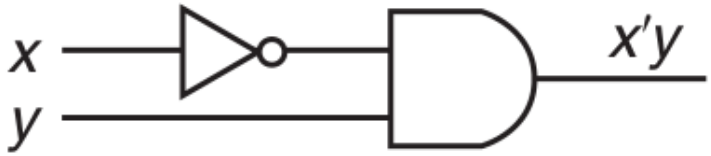

- The logic gate implementation of any function can be derived from its truth table. However, the resulting expression should be simplified to minimize the number of gates required.

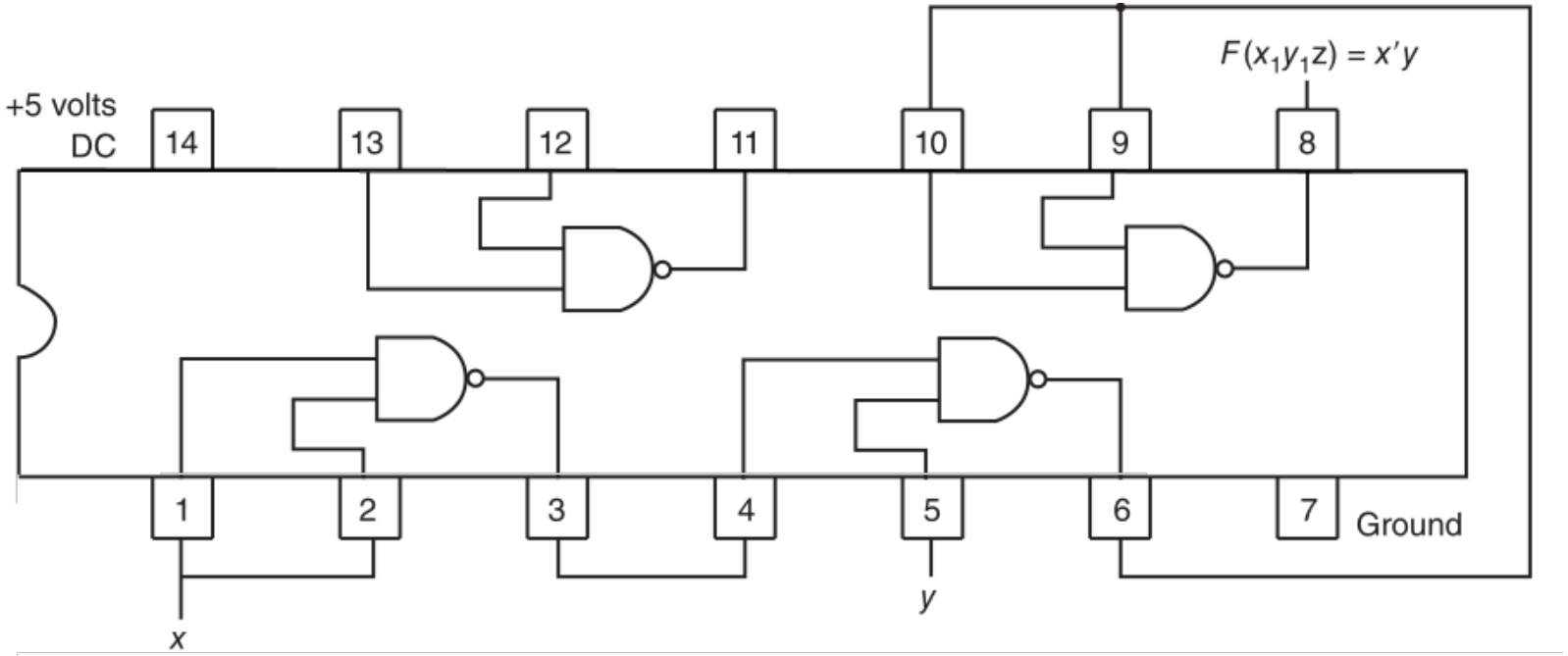
- Standard digital components are combined into single integrated circuit packages.




This is a small scale integrated circuit containing 4 NAND gates.

This chip's pins can be wired as shown below to implement the function:



Suppose we wanted to design a lighting control system that turns lights off when they are not needed.

Assume the lights are to be turned off if it is before 6 PM or if the Sun is out and it is a week day.

The decision would be based on 3 inputs:

- x = 1 if the time is before 6 PM (or 0 otherwise)
- y = 1 if the Sun is out (or 0 otherwise)
- z = 1 if it is a week day (or 0 otherwise)

The output should = 1 if the lights are to be turned off

The truth table for the lights out function is:

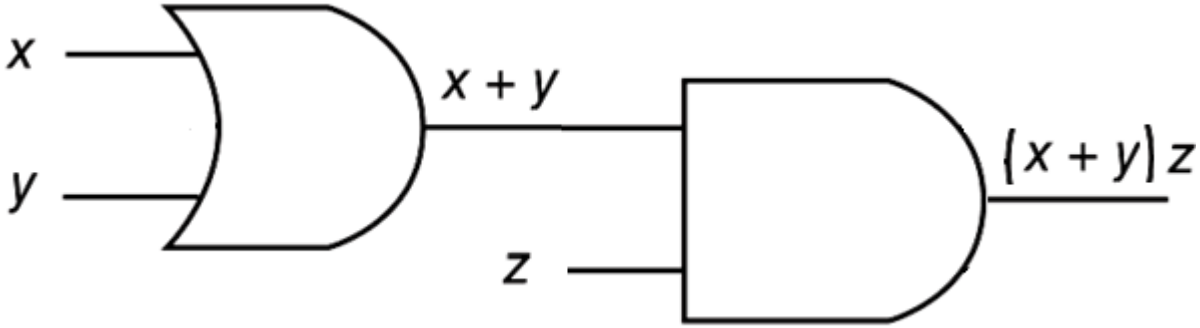
| Before 6 PM | Sun is out | Week day | Lights out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Using the logical sum of the non-zero min-terms we get:

lights out = X' Y Z + X Y' Z + X Y Z

Simplifying we get:

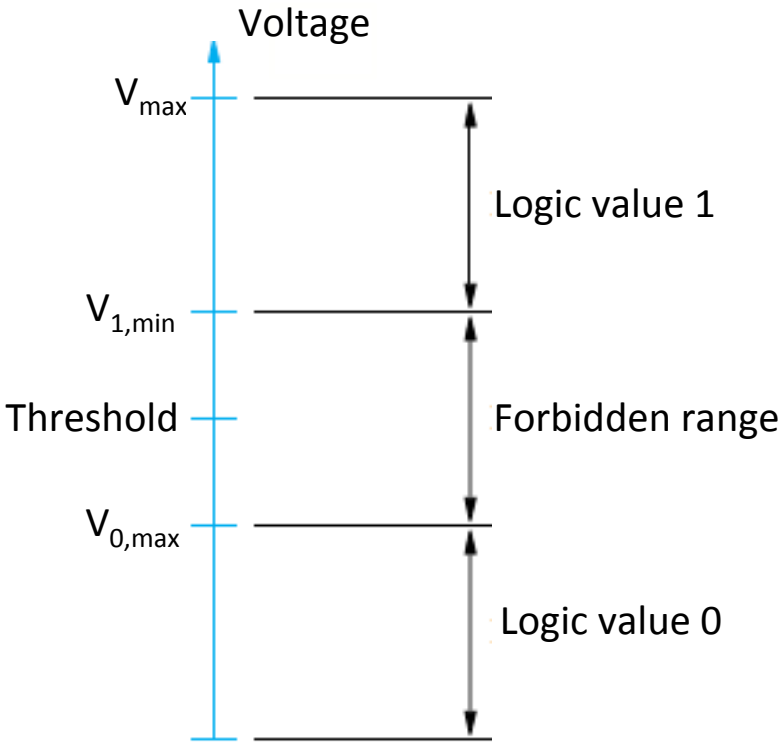
lights out = X' Y Z + X Y' Z + X Y Z = (X' Y + X Y' + X Y) Z
= ((X' Y) + X(Y' + Y)) Z = (X' Y + X) Z = (X' + X)(X+Y) Z
= (X + Y) Z



# Gate Implementation

Logic variables have values of 1 and 0, or “high” and “low”

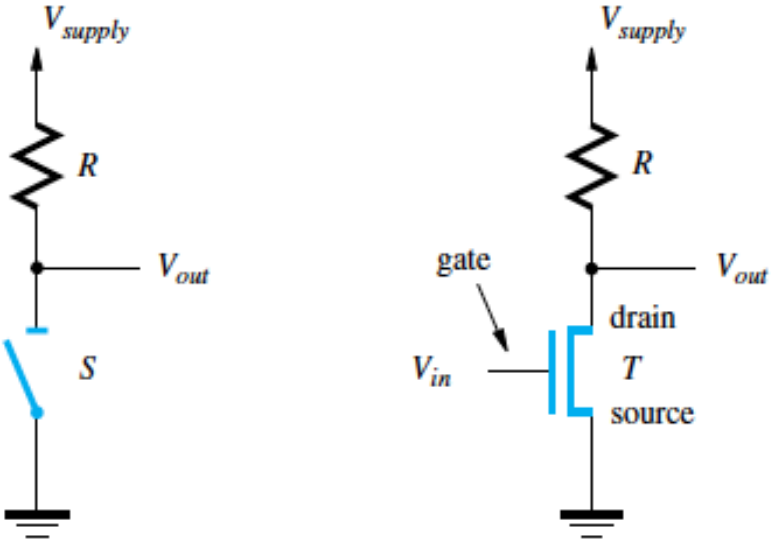
- Voltages or currents can represent logic variables
- Values above a given threshold represent one state (“on” or “off”)
- Values below the threshold represent the opposite state




Due to small random variations in the level (noise)

# NOT Gate

Digital systems use transistors as switches




Acts as an inverter

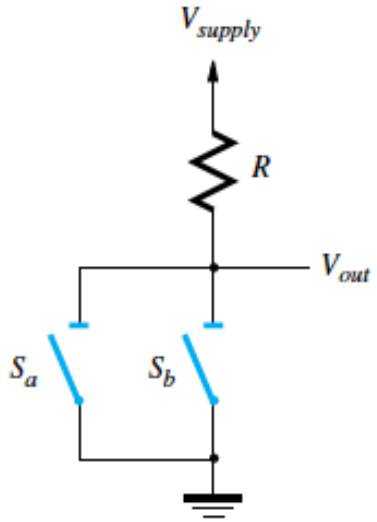
$V_{out} = V_{supply}$ (1) when the switch is open

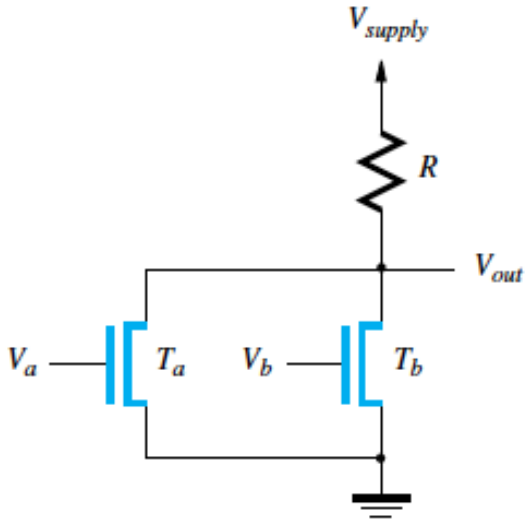
$V_{out} = 0$ when the switch is closed

Transistor act as an open switch when $V_{in} = 0$

Transistor act as a closed switch when $V_{in} = 1$

# NOR Gate




Acts as NOR Gate

| $V_a$ | $V_b$ | $V_{out}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

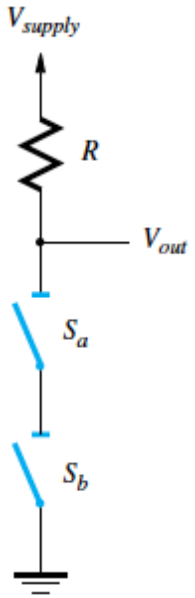
$V_{out}$ = 0 when either switch is closed

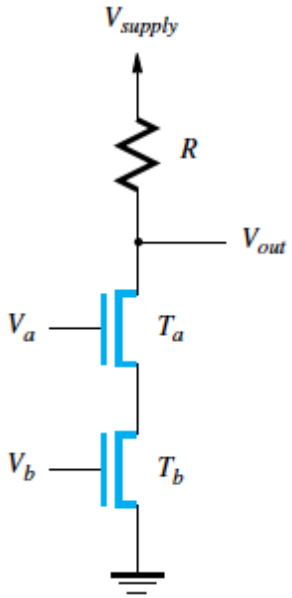
$V_{out}$ = 1 when both switches are open

Transistor act as an open switch when $V_{in}$ = 0

Transistor act as a closed switch when $V_{in}$ = 1

# NAND Gate




Acts as NAND Gate

| $V_a$ | $V_b$ | $V_{out}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

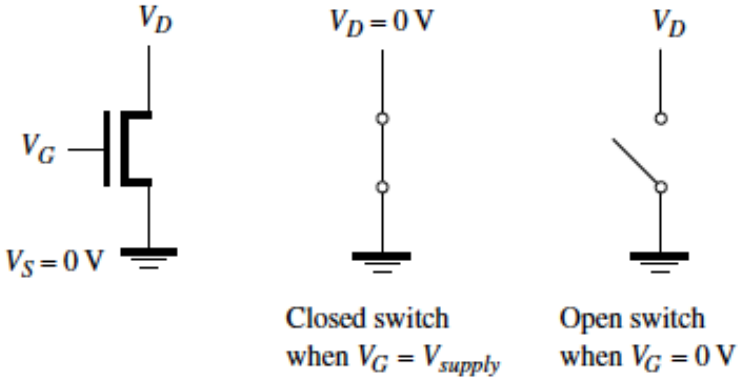
$V_{out}$ = 1 when either switch is open

$V_{out}$ = 0 when both switches are closed

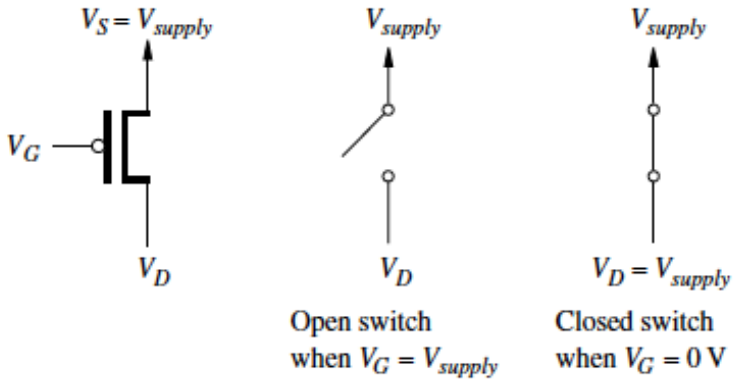
AND and OR can be implemented from NAND and NOR gates

- Just use NOT gate to invert output of NAND or NOR gates
- So more transistors are required for AND and OR gates

# NMOS & PMOS Circuits




NMOS Transistor




PMOS Transistor

Two types of metal-oxide semiconductor (MOS) are available

- NMOS-type behave as closed switch when the gate voltage is high
- PMOS-type behave as closed switch when the gate voltage is low
- Inversion bubble on input denotes PMOS

Source for NMOS transistor is connected to ground (0)

Source for PMOS transistor is connected to $V_{supply}$ (1)

# CMOS Circuits

When the switches are closed, current flows

- Power is consumed by current flowing through the resistor
- More heat results when more power is dissipated

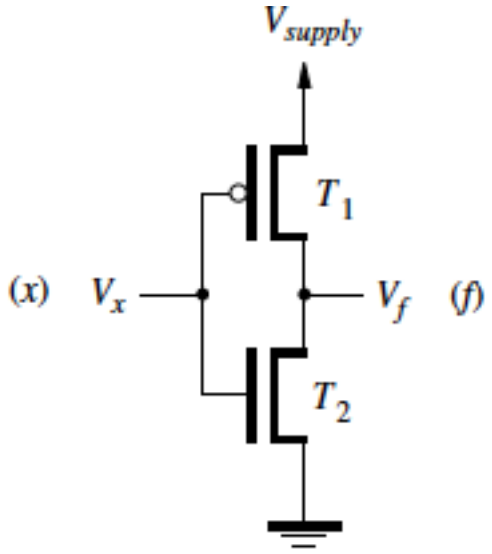
CMOS circuits combine NMOS and PMOS transistors

- CMOS means complementary metal-oxide semiconductor
- CMOS circuits consume less power

MOS transistors occupy a very small area on IC chips

- Billions can fit on a single integrated circuit (IC) chip
- Smaller transistors can switch at extremely high rates (GHz range)

# CMOS NOT Gate




Circuit

( $V_f$ = NOT $V_x$ )

| $x$ | $V_x$ | $T_1$ | $T_2$ | $V_f$ | $f$ |
|---|---|---|---|---|---|
| 0 | low | on | off | high | 1 |
| 1 | high | off | on | low | 0 |

Truth table and transistor states

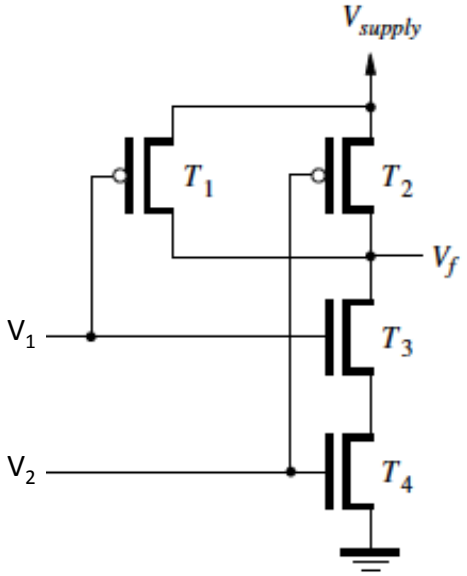
$V_x$=0 closes $T_1$ and opens $T_2$, pulling $V_f$ up to $V_{supply}$

$V_x$=1 closes $T_2$ and opens $T_1$, pulling $V_f$ down to 0

So $V_f$ is the complement of $V_x$

$T_1$ and $T_2$ operate in a complementary fashion

# CMOS NAND Gate




Circuit

( $V_f = V_1$ NAND $V_2$ )

| $V_1$ | $V_2$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $f$ |
|---|---|---|---|---|---|---|
| 0 | 0 | on | on | off | off | 1 |
| 0 | 1 | on | off | off | on | 1 |
| 1 | 0 | off | on | on | off | 1 |
| 1 | 1 | off | off | on | on | 0 |

Truth table and transistor states

If $T_1$ or $T_2$ is closed $V_f$ is pulled up to $V_{supply}$

$V_1 = 0$ closes $T_1$; $V_2 = 0$ closes $T_2$

If $T_3$ and $T_4$ are closed $V_f$ is pulled down to 0

$V_1 = 1$ closes $T_1$; $V_2 = 1$ closes $T_2$

Dissipates power only when switching

# Combinational Circuits

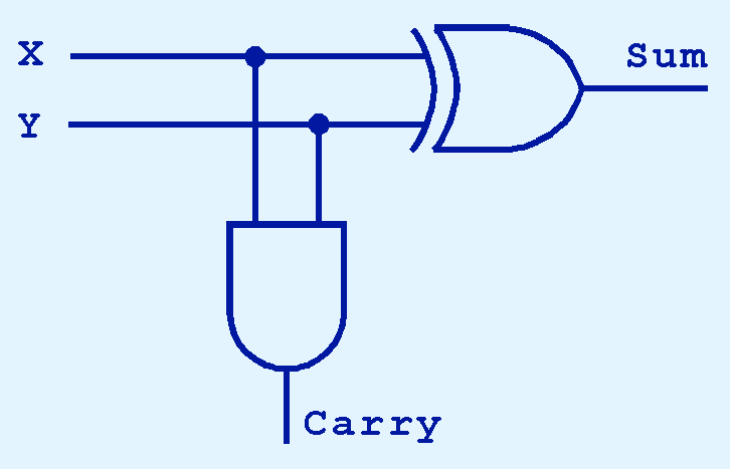
Combinational circuits are those whose output depend only on the current inputs.

As soon as the inputs change, the output changes (after a short propagational delay)

An example is a half adder :

| Inputs | | Outputs | |
|---|---|---|---|
| X | Y | Sum | Carry |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

- The sum matches the XOR operation and the carry matches the AND operation.



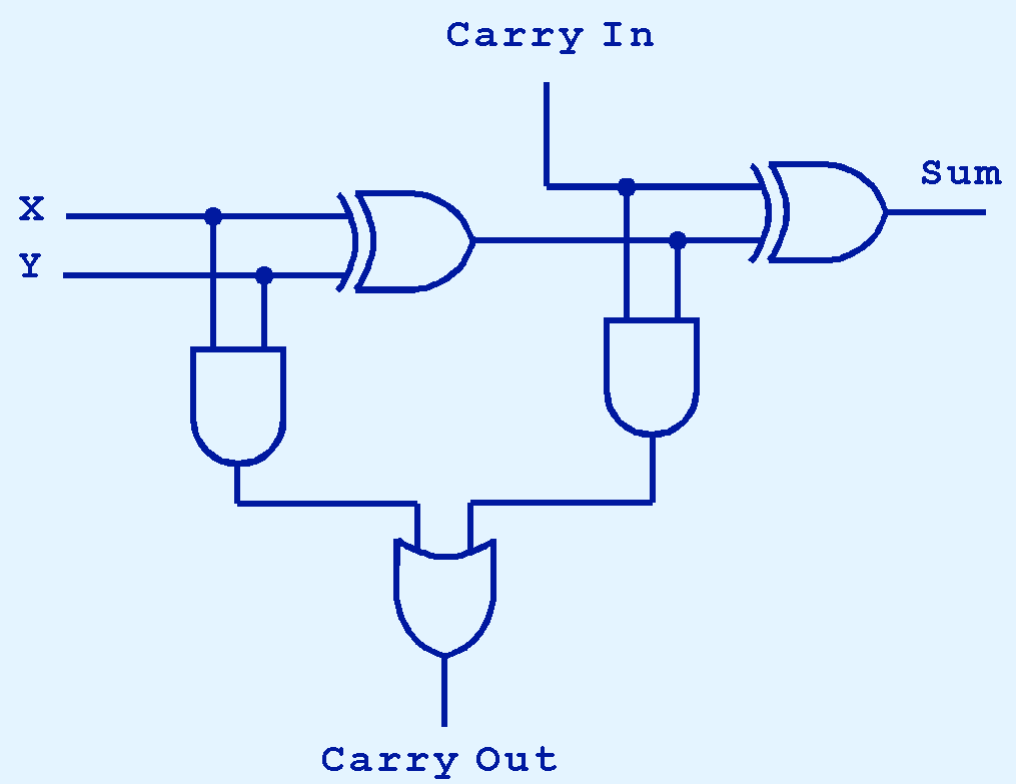

| Inputs | | Outputs | |
|---|---|---|---|
| X | Y | Sum | Carry |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

This only works for the LSB of the sum.

- The half adder becomes a full adder by including gates for processing the carry bit.
- The truth table for a full adder is shown at the right.

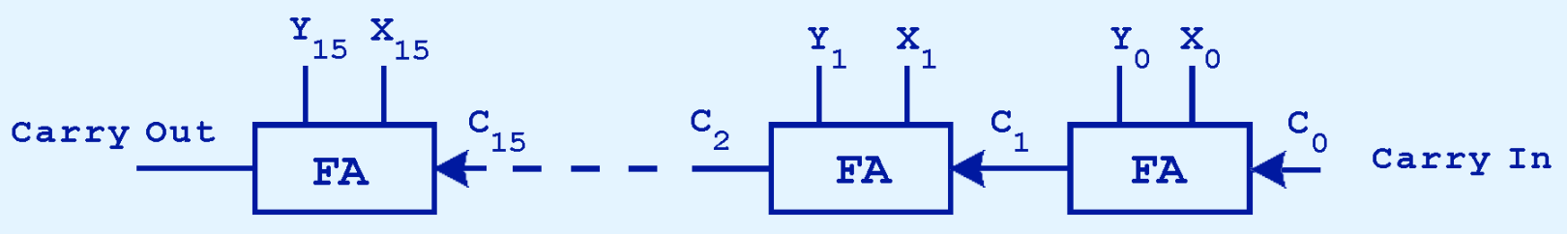
| Inputs | | | Outputs | |
|---|---|---|---|---|
| X | Y | Carry In | Sum | Carry Out |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

# Here's the completed full adder that works for any bit in the sum:



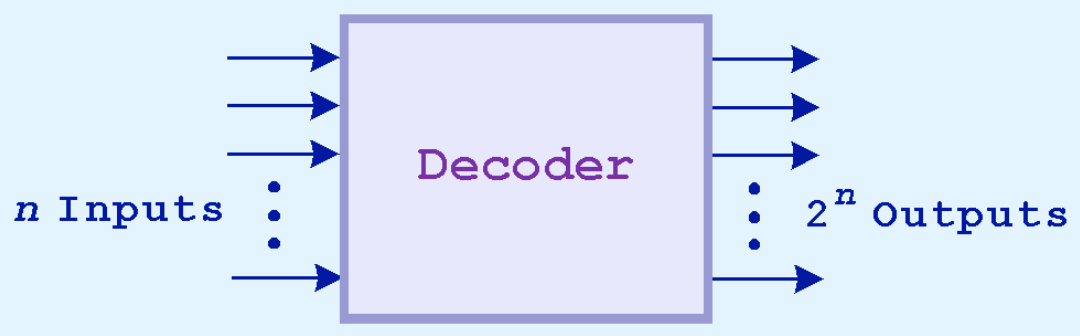

| Inputs | | | Outputs | |
|---|---|---|---|---|
| X | Y | Carry In | Sum | Carry Out |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

- Adders of any desired size can be produced by connecting full adders in series.
- The carry bit "ripples" from one adder to the next; hence, this configuration is called a *ripple-carry adder*.

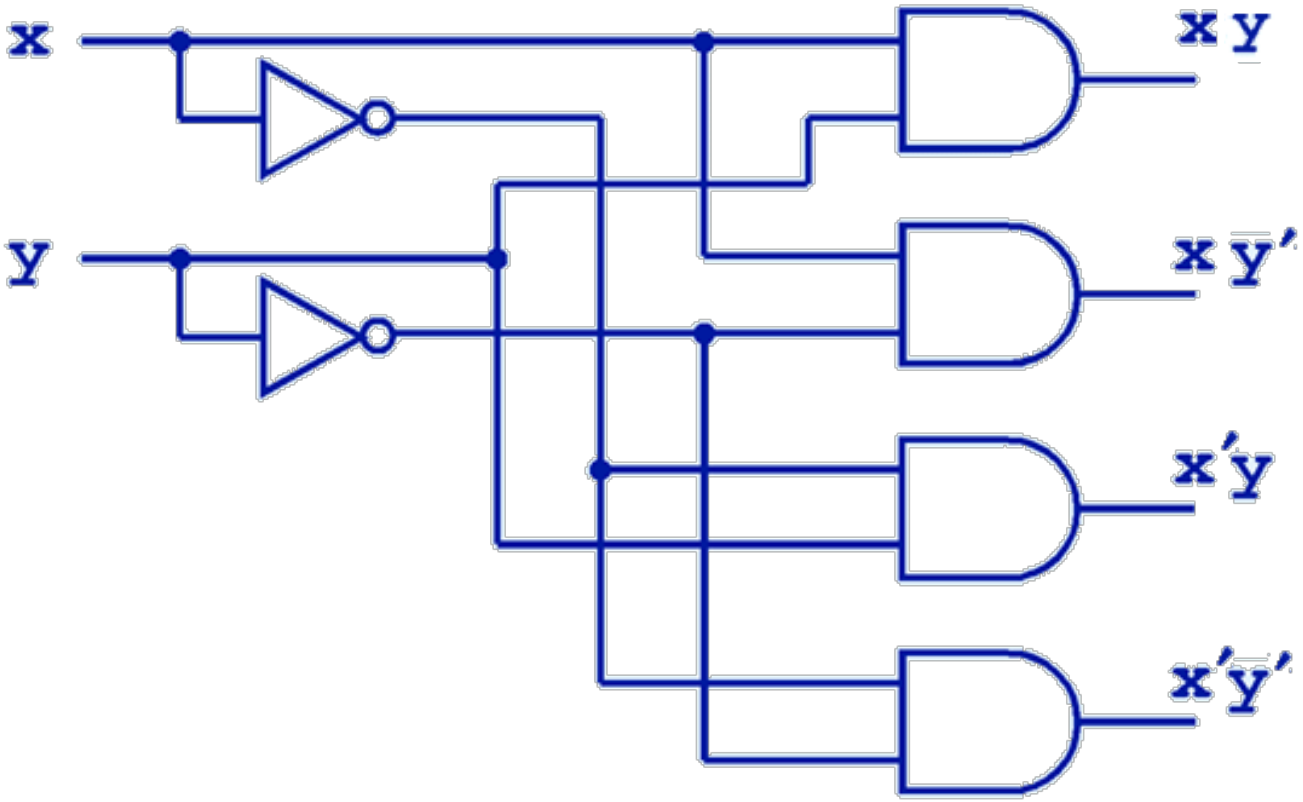



- The full adders must operate sequentially.
- A look-ahead carry adder would be more efficient because it allows the bits in the sum to be computed in parallel.

- Decoders are another important type of combinational circuit.
- Among other things, they are useful in selecting a memory location indicated by a binary value placed on the address lines of a memory bus.
- Address decoders with *n* inputs can select any of $2^n$ locations.

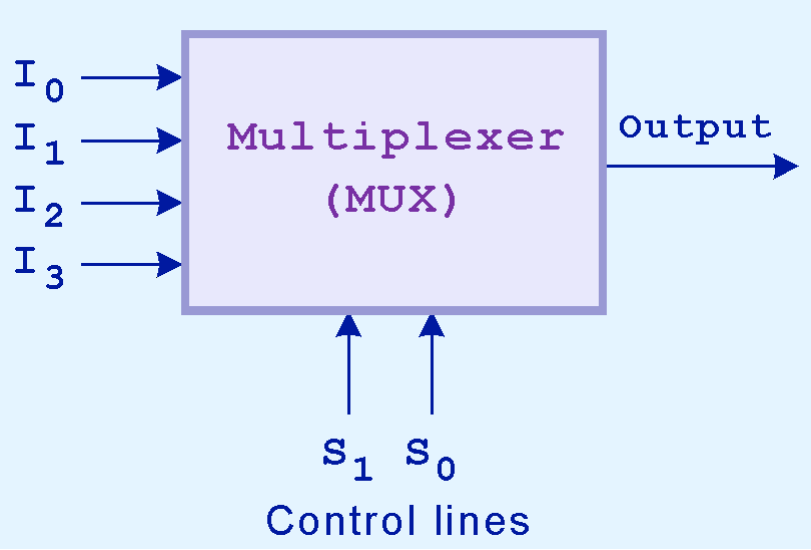


- A 2-to-4 decoder could be implemented as:



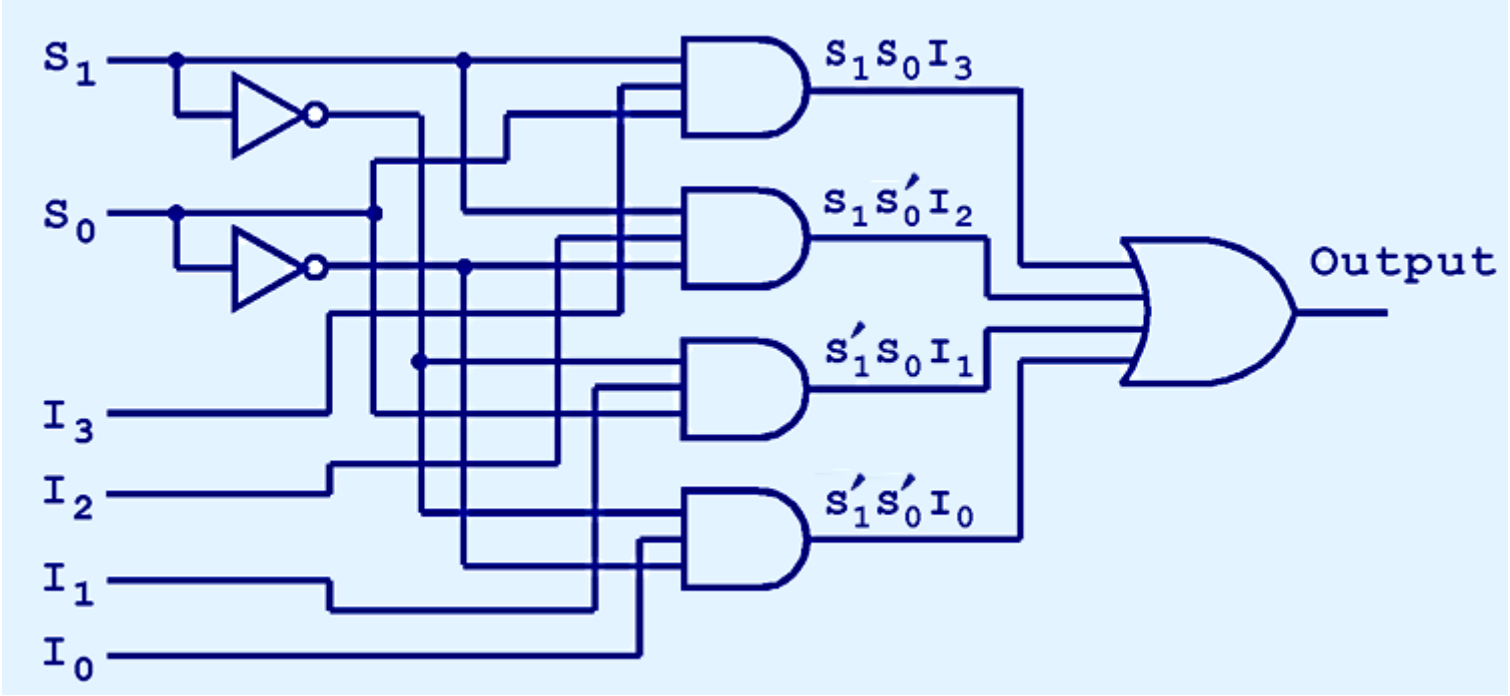

The 2-bit number xy is decoded to select one of 4 outputs.

- A multiplexer does just the opposite of a decoder.
- It selects a single output from several inputs.
- The particular input chosen for output is determined by the value of the multiplexer's control lines.
- To be able to select among *n* inputs, $\log_2 n$ control lines are needed.




**Multiplexers are also called selectors.**

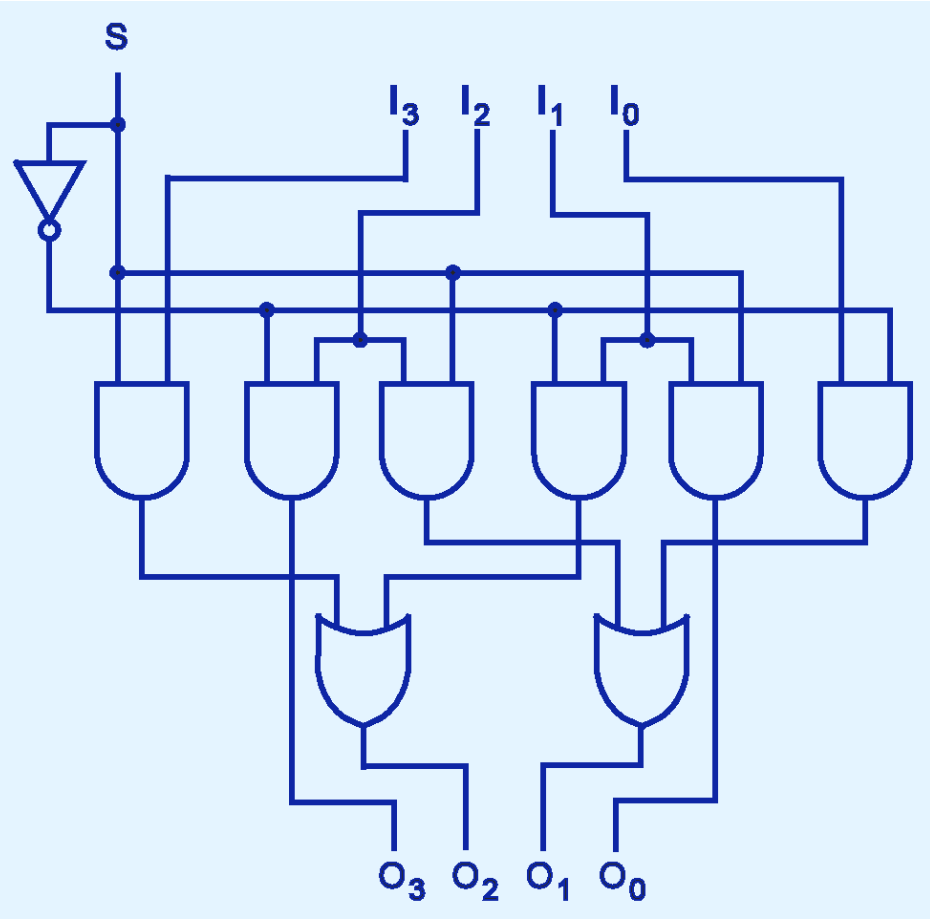
- A possible implementation of a 4-to-1 multiplexer is shown below:




$S_1S_0$ selects one of 4 inputs $I_3$, $I_2$ , $I_1$ or $I_0$ to pass to the output.

- Multiplication and division involve a series of additions, subtractions and shifting operations.
- We have seen how logic gates can add numbers
- Once we see how to use logic gates to perform shifing, we can then compute products, quotients and remainders.
- A shifter moves the bits within a binary pattern one position to the left or right.

One way to implement a shifter is shown below:




S = 0 shifts left, S=1 shifts right.

- Combinational logic circuits generate outputs that depend only on the current set of inputs.
- *Sequential logic circuits* generate outputs that depend on the previous history of inputs.
  - These circuits have to "remember" their current state.
- *Sequential logic circuits* are used to implement devices such as registers and memories.
- These memory-type devices will be examined later in the course.

# Boolean Algebra

Digital computers contain circuits that implement Boolean functions.

Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values ("true" and "false")

Boolean expressions are created by performing operations on Boolean variables.

Common Boolean operators include AND, OR, and NOT.

# Boolean Algebra

In digital systems, these values are “on” and “off,” 1 and 0, or “high” and “low.”

This is why the binary numbering system is a natural basis for digital systems.

Logic circuits take one or more Boolean or digital inputs and generate a digital output.

We will examine how logic circuits can be made up from combinations of basic logic gates.

# Boolean Algebra

A Boolean function has:

- At least one Boolean variable,
- At least one Boolean operator, and
- At least one input from the set {0,1}.

It produces an output that is also a member of the set {0,1}.

Boolean functions can be represented using truth tables and can be implemented using logic gates

The truth table for the Boolean function:

$$F(x,y,z) = xz' + y$$

is shown at the right.

The extra (shaded) columns hold evaluations of subparts of the function.

$$F(x,y,z) = xz' + y$$

| x | y | z | z' | xz' | xz' + 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 |

Like arithmetic operators, Boolean operators have rules of precedence.

The NOT operator has highest priority, followed by AND and then OR

z′ denotes NOT z

xz denotes x AND y

x + y denotes x OR y

$F(x, y, z) = xz' + y$

| x | y | z | z' | xz' | xz' + 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 |

- The simpler the Boolean functions, the fewer the number of logic gates needed to implement them.

- Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
  - Best to reduce our Boolean functions to their simplest form.

- There are a number of Boolean identities that help us to do this.

Most Boolean identities have an AND (product) form as well as an OR (sum) form. We give our identities using both forms. The group below is rather intuitive:

| Identity Name | AND Form | OR Form |
|---|---|---|
| Identity Law | $1x = x$ | $0 + x = x$ |
| Null Law | $0x = 0$ | $1 + x = 1$ |
| Idempotent Law | $xx = x$ | $x + x = x$ |
| Inverse Law | $xx' = 0$ | $x + x' = 1$ |

This second group of Boolean identities should be familiar to you from your study of algebra:

| Identity Name | AND Form | OR Form |
|---|---|---|
| Commutative Law | $xy = yx$ | $x+y = y+x$ |
| Associative Law | $(xy)z = x(yz)$ | $(x+y)+z = x + (y+z)$ |
| Distributive Law | $x+yz = (x+y)(x+z)$ | $x(y+z) = xy+xz$ |

The next group of Boolean identities are perhaps the most useful.

If you have studied set theory or formal logic, these laws are also familiar to you.

| Identity Name | AND Form | OR Form |
|---|---|---|
| Absorption Law | $x(x+y) = x$ | $x + xy = x$ |
| DeMorgan's Law | $(xy)' = x' + y'$ | $(x+y)' = x'y'$ |
| Double Complement Law | $(x)'' = x$ | |

The example below illustrates the use of Boolean identities to simplify a logic expression:

```
F(x,y,z) = xy + x'z + yz
```

$$
\begin{aligned}
F(x,y,z) &= xy + x'z + yz \\
&= xy + x'z + yz(1) && \text{(Identity)} \\
&= xy + x'z + yz(x + x') && \text{(Inverse)} \\
&= xy + x'z + (yz)x + (yz)x' && \text{(Distributive)} \\
&= xy + x'z + x(yz) + x'(zy) && \text{(Commutative)} \\
&= xy + x'z + (xy)z + (x'z)y && \text{(Associative twice)} \\
&= xy + (xy)z + x'z + (x'z)y && \text{(Commutative)} \\
&= xy(1 + z) + x'z(1 + y) && \text{(Distributive)} \\
&= xy(1) + x'z(1) && \text{(Null)} \\
&= xy + x'z && \text{(Identity)}
\end{aligned}
$$

- A Boolean expression may have multiple logically equivalent (or “synonymous”) forms

- There are two canonical (i.e. standardized) forms for Boolean expressions:
  - sum-of-products and product-of-sums.

- The Boolean product is the AND operation and the Boolean sum is the OR operation.

In the sum-of-products form, ANDed variables are ORed together.

For example:

$$F(x,y,z) = xy + xz + yz$$

In the product-of-sums form, ORed variables are ANDed together:

For example:

$$F(x,y,z) = (x+y)\,(x+z)\,(y+z)$$

To obtain the sum-of-products form using its truth table, we list the values of the variables that result in a true function value (=1).

Each group of variables is then ORed together.

$$F(x,y,z) = (x'yz') + (x'yz) + (xy'z') + (xyz') + (xyz)$$

Use the complement of the 0 inputs. The result is the logical sum of the non-zero “minterms”.

$$F(x,y,z) = xz' + y$$

| x | y | z | xz'+ 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 |

To obtain the product-of-sums form using its truth table, we list the values of the variables that result in a false function value (=0).

Each group of variables is then ANDed together.

$$F = (x+y+z)(x+y+z')(x'+y+z')$$

Use the complement of the 1 inputs. The result is the logical product of the Zero “maxterms”.

$F(x,y,z) = xz' + y$

| x | y | z | xz'+ 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 |

It can be more economical to build a circuit using the complement of a function (and complementing its result) than 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:

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

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 that the complement of:

$$F(x,y,z) = (xy) + (x'y) + (xz')$$

is:

$$\begin{aligned} F'(x,y,z) &= ((xy) + (x'y) + (xz'))' \\ &= (xy)'(x'y)'(xz')' \\ &= (x' + y')(x + y')(x' + z) \end{aligned}$$

This concludes our examination of Boolean Algebra as a basis for describing the behavior of logic circuits.

Next we will see how logic gates are used to implement various functions required within the computer.