



Lecture 11

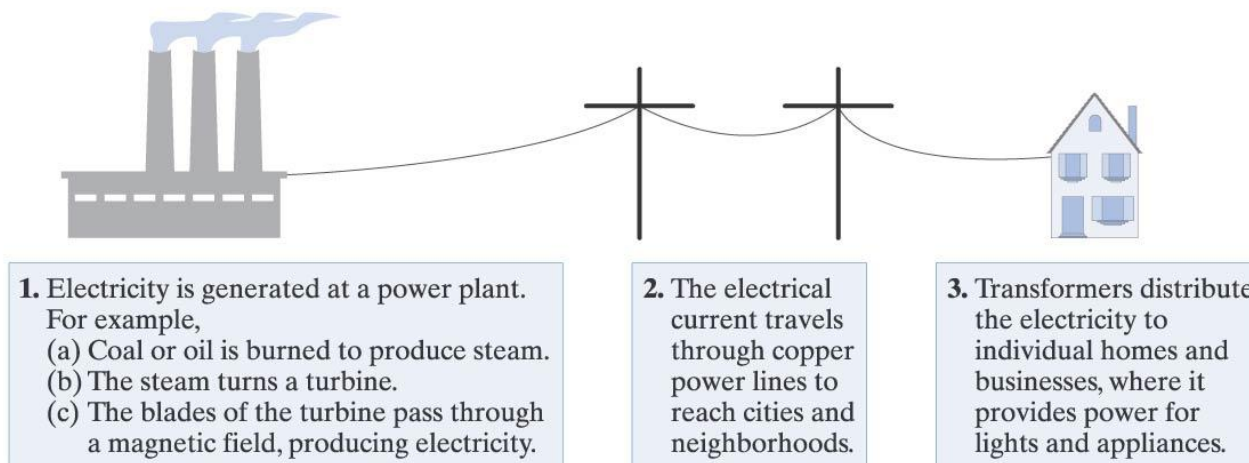
Inside the Computer – Transistors and Integrated Circuits

Electricity and Switches

- modern computers are powered by electricity, using electrical signals to store and manipulate information
- the components of a computer require electrical power to carry out their assigned task
 - electricity generates the light that shines through a computer screen, illuminating the individual pixels that make up images and letters
 - electricity runs the motor that spins the hard-drive disk, allowing information to be accessed
 - main memory and CPU employ electrical signals to store and manipulate data
 - bit patterns are represented by the presence or absence of electrical current along a wire

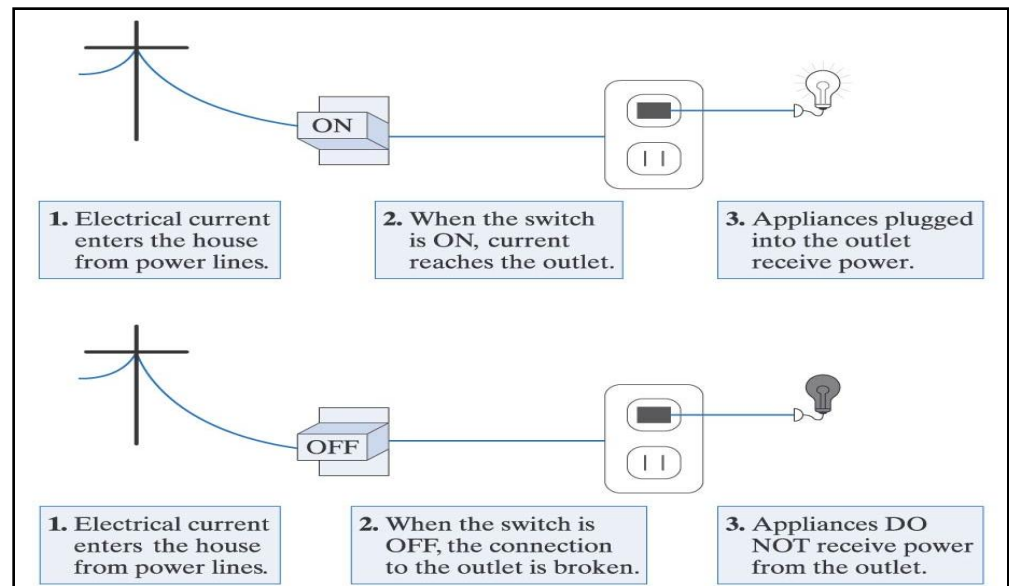
Electricity Basics

- electricity is a flow of **electrons**, the negatively charged particles in atoms, through a medium
 - good conductors of electricity allow for the flow of electrons with little resistance (e.g., copper, silver, gold)
 - other elements, especially nonmetals, are poor conductors (e.g., carbon, oxygen)
- electricity can be quantified in **amperes** or **voltage**
 - amperes gauge electron flow: 1 amp is equal to 6.24 quintillion electrons flowing past a given point each second
 - voltage measures the physical force produced by the flow of electrons: standard household in United States has 110 to 120 volt outlets



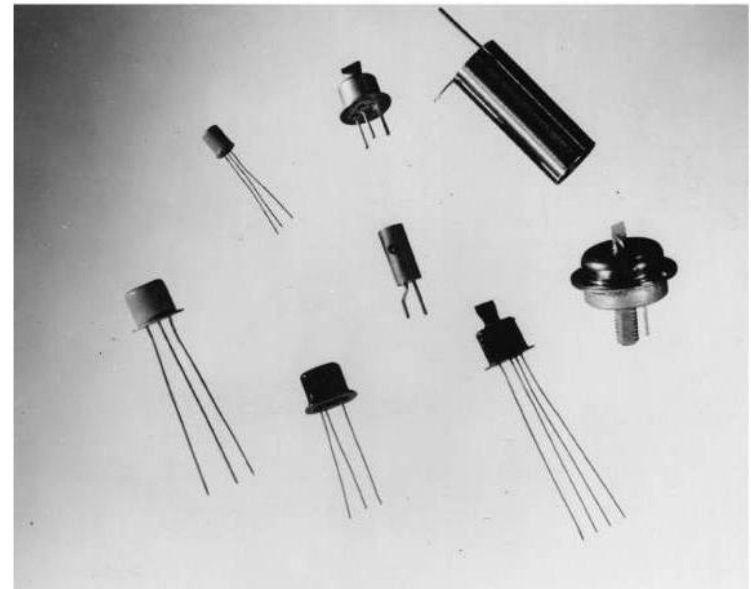
Switches

- the most basic tool for controlling the flow of electricity is a **switch**
 - a switch can be flipped to connect or disconnect two wires, thus regulating the flow of electricity between them
- example:* a light switch on a wall serves as an intermediary between the power line entering your home and the outlet that operates a lighting fixture
 - if the switch is turned on, then the wires that link the outlet to the power line are connected, and the lighting fixture receives electricity
 - if the switch is turned off, then the connection is interrupted, and no power reaches the outlet



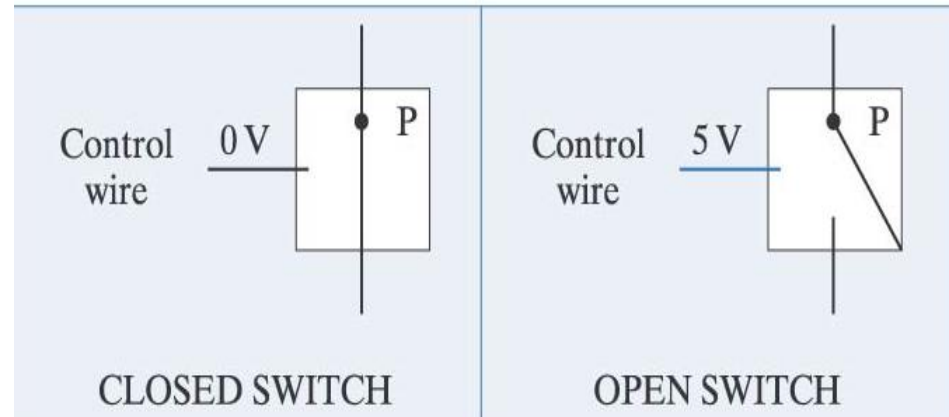
Transistors

- as we saw in Chapter 6, advances in switching technology have defined the generations of computers
 - 1930's – electromagnetic relays served as physical switches, whose on/off positions were controlled by the voltage to a magnet
 - 1940's – vacuum tubes replaced relays, which were faster (since no moving parts) but tended to overheat and burn out frequently
 - 1948 – the transistor was developed by Bardeen, Brattain, and Shickley
 - a transistor is a solid piece of metal attached to a wire that serves as a switch by alternatively conducting or resisting electricity
 - transistors allowed for the development of smaller, faster machines at a lower cost
- *semiconductors* are metals that can be manipulated to be either good or bad conductors of electricity
 - the first transistors were made of germanium and gold, but modern transistors are constructed from silicon
 - through a process known as *doping*, impurities are added to a slab of silicon, causing the metal to act as an electrical switch

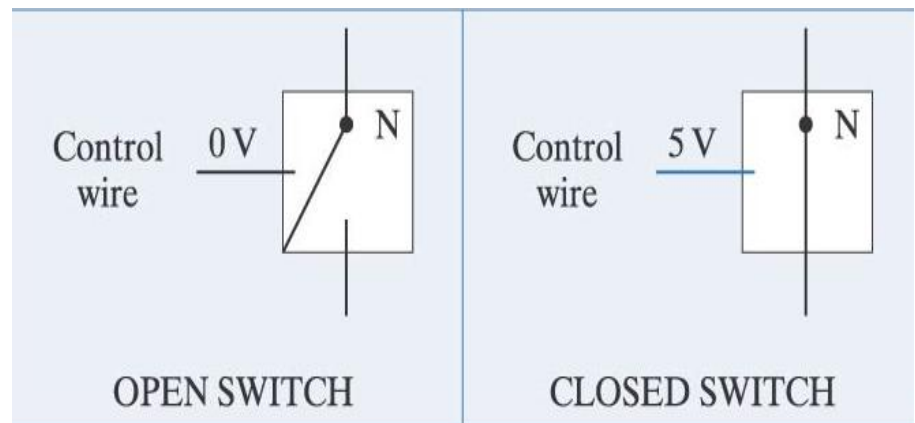


Transistors as Switches

- a PMOS transistor is positively doped, so that the switch is "closed" when there is no current on the control wire, but "opens" when current is applied



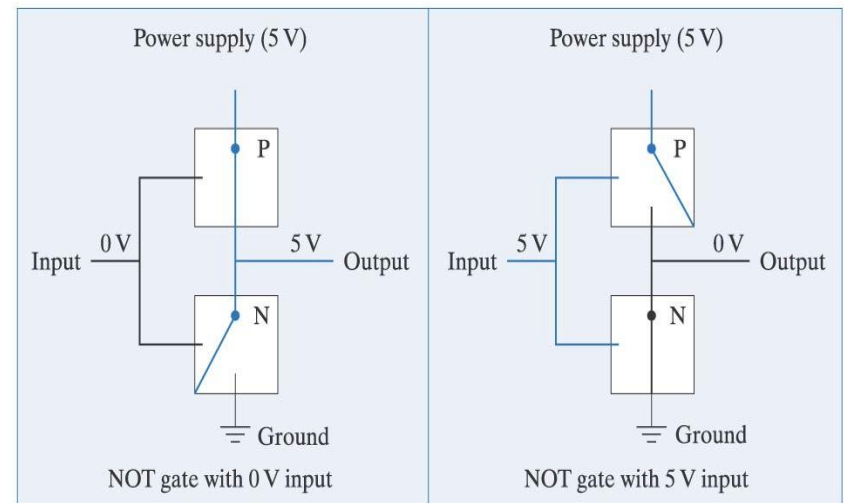
- an NMOS transistor is negatively doped, so that the switch is "open" when there is no current, but "closes" when there is current



From Transistors to Gates

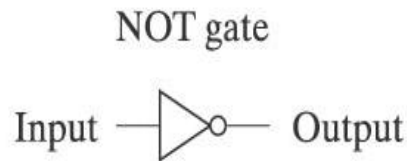
7

- transistors can be combined to form a circuit, which controls the flow of electricity in order to produce a particular behavior
- *example*: consider the following circuit combining two transistors
 - if no current (0 volts) is applied to the input wire, the PMOS transistor will close to allow current to travel on the output wire, and the NMOS transistor will open to disconnect the ground
 - if current (5 volts) is applied to the input wire, the PMOS transistor will open to disconnect the output wire, and the NMOS transistor will close to ground the input
 - the end result is that the output is the opposite of the input
 - this circuit known as a NOT gate



Gates and Binary Logic

- the term “gate” suggests a simple circuit that controls the flow of electricity
 - in the case of a NOT gate, the flow of electricity is manipulated so that the output signal is always opposite of the input signal
 - we can think of a gate as computing a function of binary values



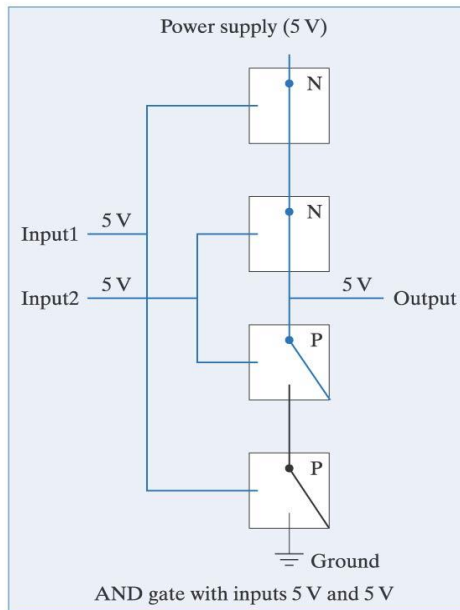
input	NOT output
0	1
1	0

note: NOT gates invert voltages in the same way that the JavaScript NOT operator (!) inverts Boolean values

- 0 corresponds to false; 1 corresponds to true
- the symbol to the left (triangle w/ circle) is often used to denote a NOT gate
- the *truth table* to the right describes the mapping of input to output

Gates and Binary Logic

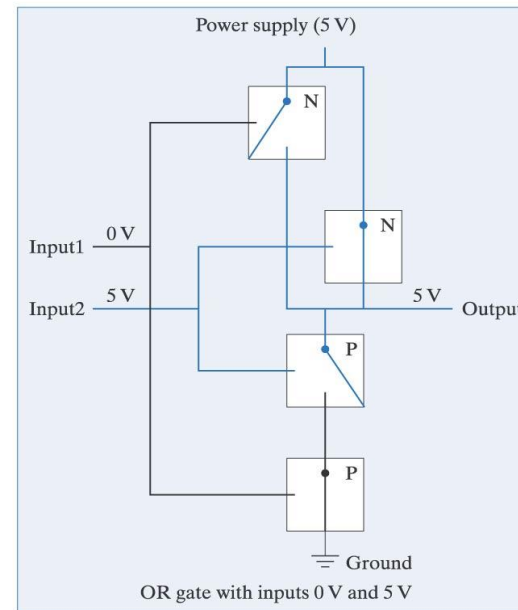
- many other simple circuits can be defined to perform useful tasks
 - AND gate** – produces voltage on its output wire if both input wires carry voltage
 - OR gate** – produces voltage on its output wire if either input wire carries voltage



AND gate

Input1 Input2 Output

input1	input2	AND output
0	0	0
0	1	0
1	0	0
1	1	1



OR gate

Input1 Input2 Output

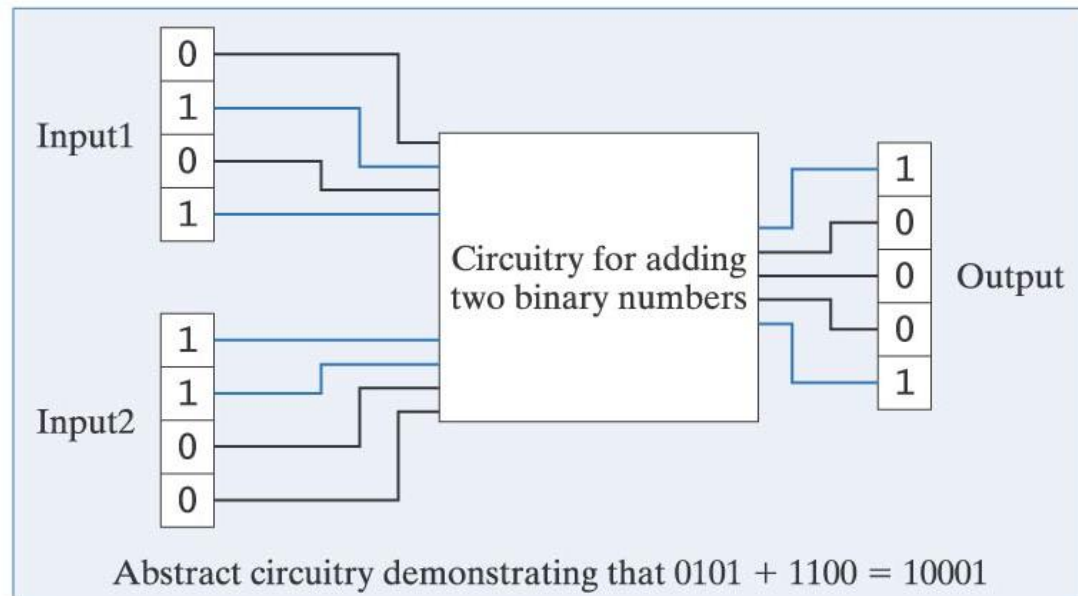
input1	input2	OR output
0	0	0
0	1	1
1	0	1
1	1	1

- AND, OR, and NOT gates can be combined to construct all the circuitry required to store and manipulate information within a computer

From Gates to Circuits

10

- transistors are connected to form basic logic gates, which are then combined to build more advanced circuitry
- example: adding two binary numbers
 - we can represent a 4-bit binary number using 4 wires
 - current on a wire signifies a 1 bit for that place; no current signifies 0



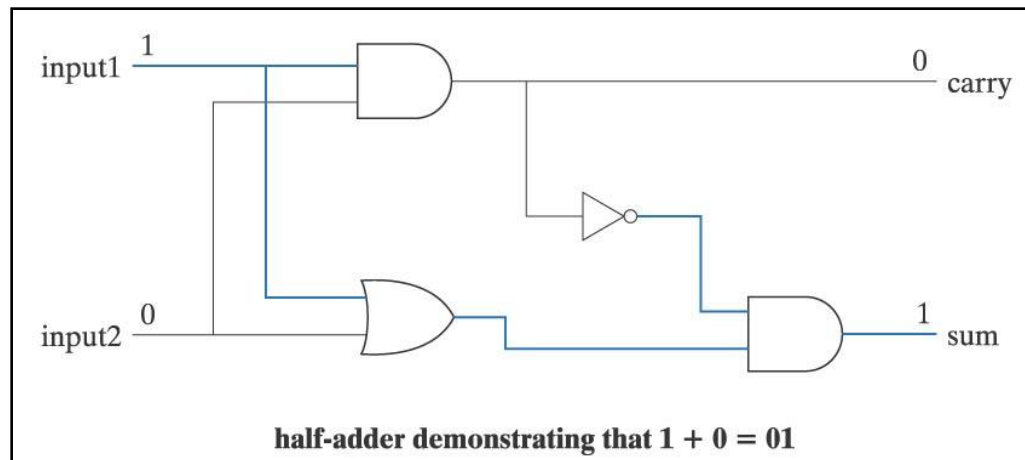
Half-adder Circuit

- recall the rules of binary addition

$$\begin{array}{r} 1 \\ 112 \\ + 12 \\ \hline 102 \end{array}$$

$$\begin{array}{r} 1 \\ 102 \\ + 112 \\ \hline 1012 \end{array}$$

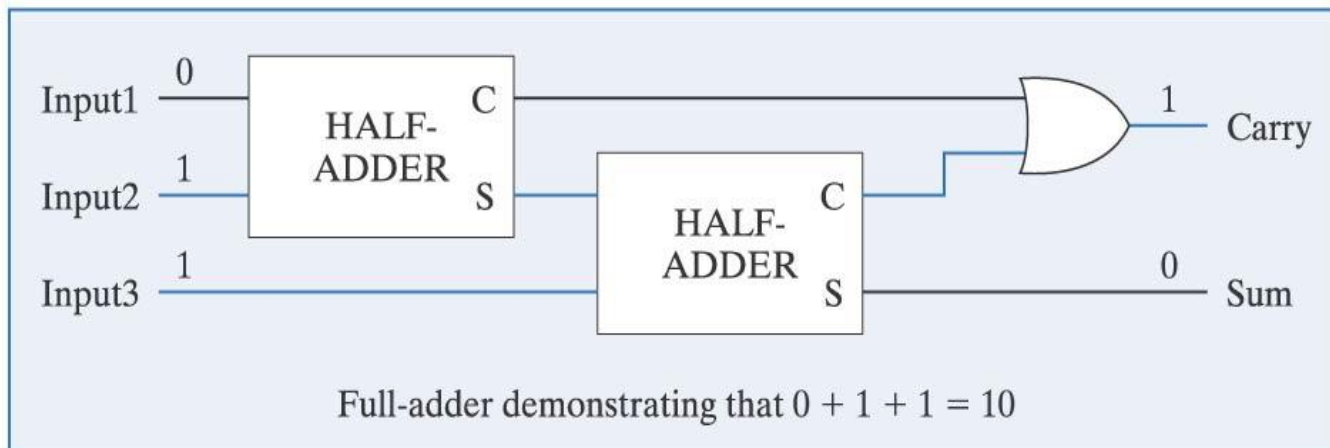
- although binary addition is relatively straightforward, designing a circuit for adding binary numbers is quite complex
 - instead of starting at the transistor level, we can use AND, OR, and NOT gates
 - focus first on the addition of 2 bits
 - requires two input lines, two output lines (sum of inputs and possible carry)
 - the circuit consist of four gates (known as a half-adder)



Full-adder Circuit

12

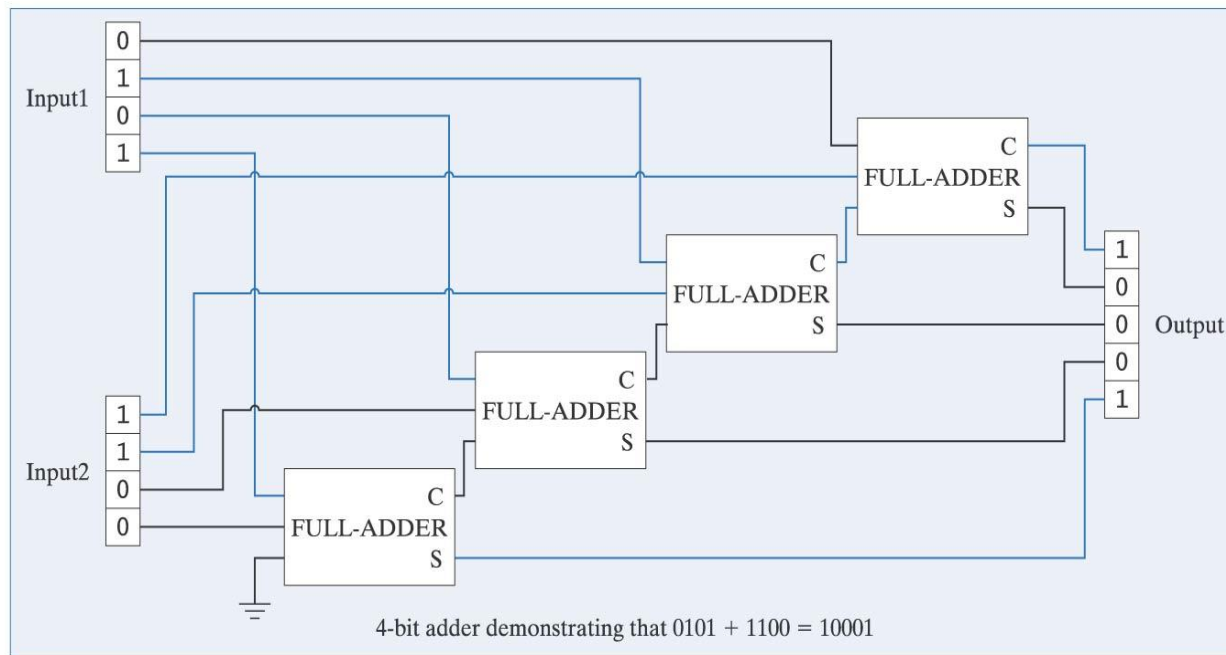
- the term “half-adder” refers to the fact that when you add binary numbers containing more than one bit, summing the corresponding bit pairs by column is only half the job
 - you must also consider that a bit might be carried over from the previous addition
 - using half-adders and logical gates as building blocks, we can design a circuit that takes this into account (known as a **full-adder**)



4-bit Adder Circuit

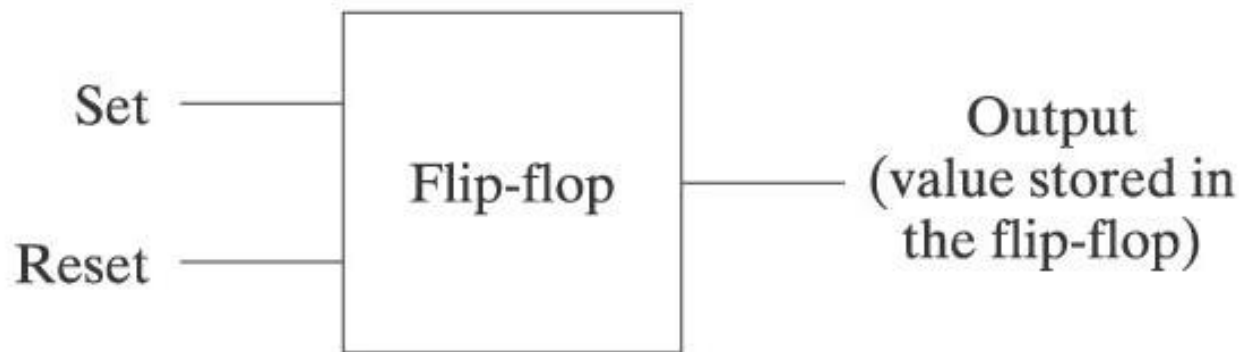
13

- using full-adders as building blocks, we can design a more complex circuit that sums two 4-bit numbers
 - since a full-adder is required to add each corresponding bit pair together (along with possible carry), the circuit will need four full-adders wired together



Designing Memory Circuitry

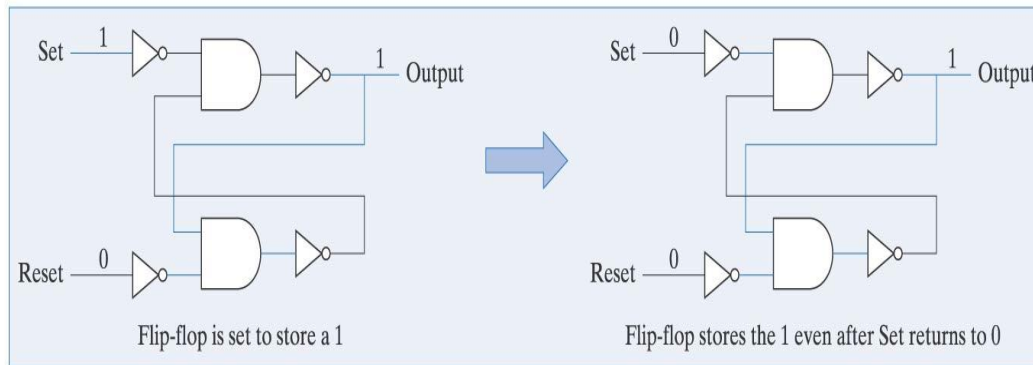
- main memory and registers within the CPU are composed of circuitry
 - whereas adders manipulate inputs to produce outputs, memory circuits must maintain values over time
 - the simplest circuit for storing a value is known as a *flip-flop*
 - it can be set to store a 1 by applying current on an input wire
 - it can be reset to store a 0 by applying current on another



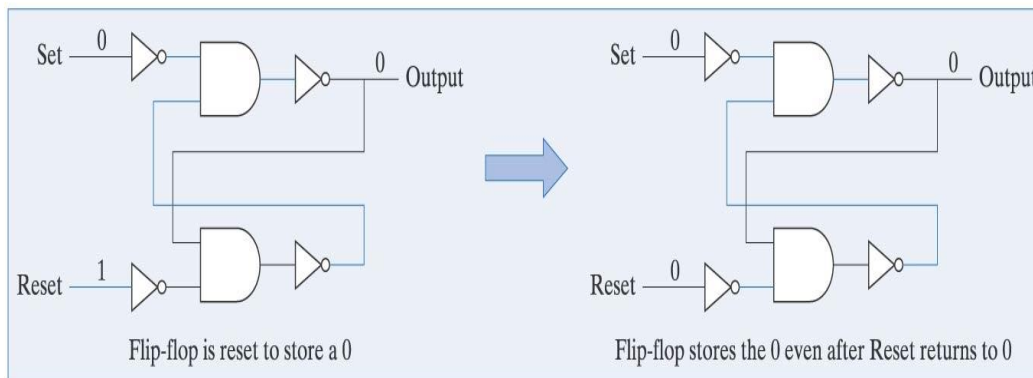
Flip-flop Circuit

15

- a **flip-flop** stores a value by feeding the output currents back into the circuit
 - the value is maintained by current flowing around and around the circuit
 - a current on the Set wire produces current on the output, which then cycles



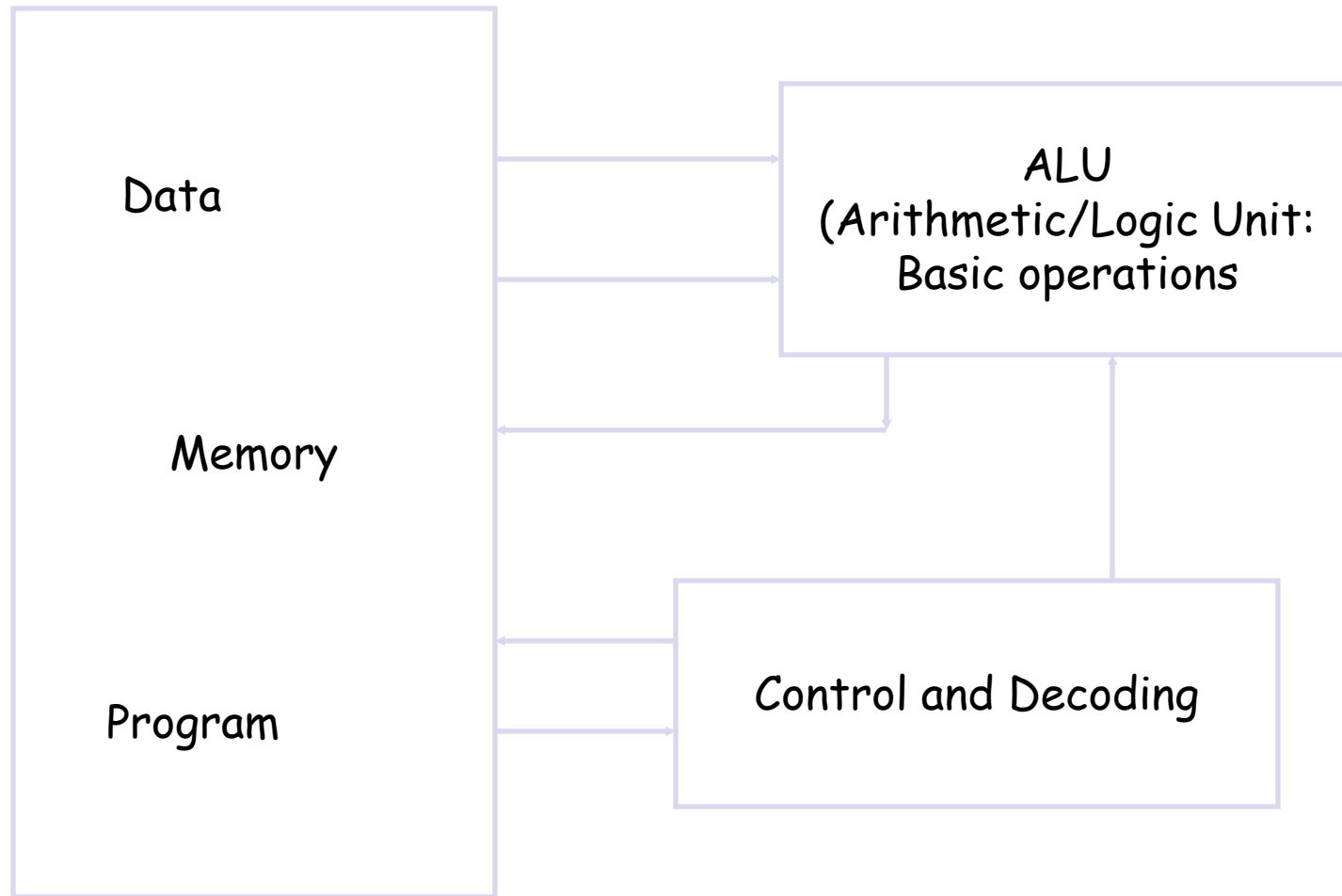
- a current on the Reset wire produces no current on the output



Summary

- Controllable Switches are easy to make
 - There are many kinds of controllable switches.
 - For example: NMOS, PMOS switches
- These switches can be used to put together to make “Logic Gates”:
 - AND, OR, NOT
- Logic Gates can be put together to make **half adder**, **full adders** and **multi-bit adders**
 - So we can see they can be used for other such circuits as well
- Logic Gates can be used to make circuits that “remember” or store data
 - flop-flop circuit
- A Computer includes the following most important units:
 - An ALU (Arithmetic Logic Unit)
 - Instruction Decoding and associated circuits
 - Memory
 - Stored Program

Components of a basic computer



Boolean Functions

- We can interpret high or low voltage as representing true or false.
- A variable whose value can be either 1 or 0 is called a **Boolean variable**.
- A **Boolean function** has one or more input Boolean variables and yields results based on the values of these variables.
- A Boolean function of n variables has 2^n possible combinations of the input value. These combinations can be expressed in an expression or a table called **a truth table**.

Simple Boolean Function -- AND

Function $f : \{0,1\}^2 \rightarrow \{0,1\}^1$

Boolean algebra notation for AND:

$$x1 \text{ AND } x2 \Leftrightarrow x1 * x2$$

$$x1 \text{ AND } x2 \Leftrightarrow x1 \bullet x2$$

$$x1 \text{ AND } x2 \Leftrightarrow x1 \ x2$$

So we can express the function as

$$f(x1, x2) = x1 * x2$$

$$f(x1, x2) = x1 \bullet x2$$

$$f(x1, x2) = x1 \ x2$$

Truth Table

x1	x2	x1 AND x2
0	0	0
0	1	0
1	0	0
1	1	1

Simple Boolean Function -- OR

Function $f : \{0,1\}^2 \rightarrow \{0,1\}^1$

Boolean algebra notation for OR:

$$x1 \text{ OR } x2 \Leftrightarrow x1+x2$$

So we can express the function as
 $f(x1, x2)=x1+x2$

Truth Table

x1	x2	x1 OR x2
0	0	0
0	1	1
1	0	1
1	1	1

Simple Boolean Function -- NOT

Function $f : \{0,1\}^1 \rightarrow \{0,1\}^1$

Boolean algebra notation for NOT:

$$\text{NOT } x \Leftrightarrow \sim x$$

$$\text{NOT } x \Leftrightarrow x'$$

$$\text{NOT } x \Leftrightarrow \bar{x}$$

So we can express the function as

$$f(x) = \sim x$$

$$f(x) = x'$$

$$f(x) = \bar{x}$$

Truth Table



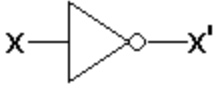
x	!x
0	1
1	0

Precedence

- Precedence are important, but not too difficult.
 - NOT has the highest precedence, followed by AND, and then OR.
 - For example, the following two functions would generate different results due to the order of operations performed.
 - $f(x,y,z) = x + y' z + x'$
 - $g(x,y,z) = (x + y') z + x'$

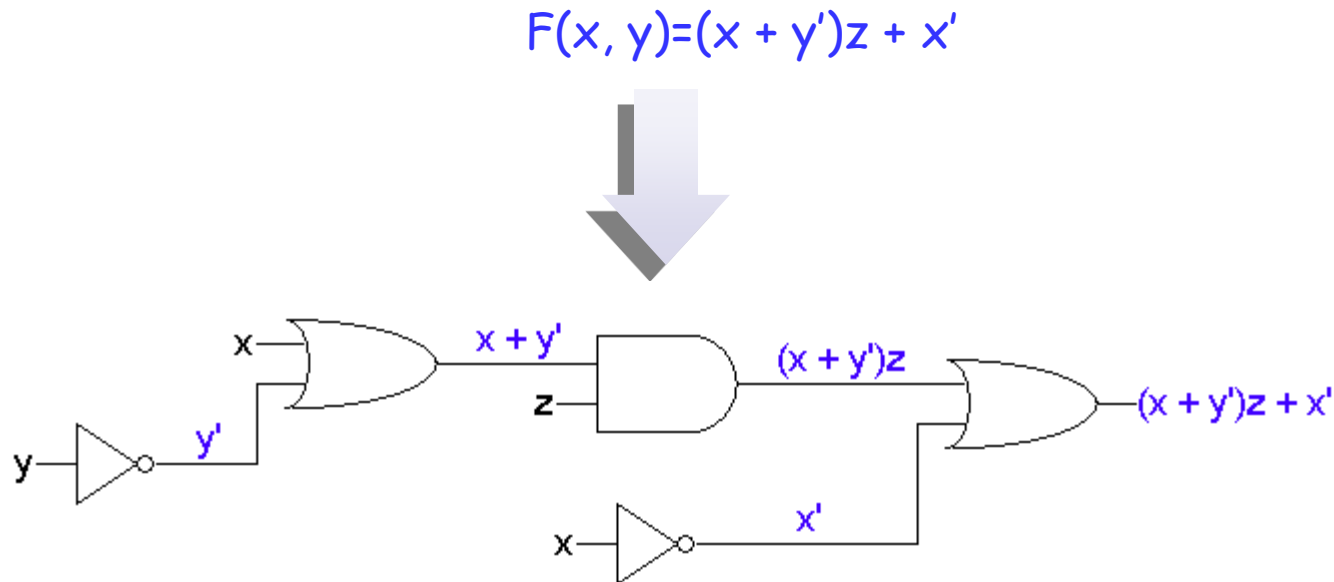
Primitive logic gates

- Each of our basic operations can be implemented in hardware using a **primitive logic gate**.
 - Symbols for each of the logic gates are shown below.
 - These gates output the product, sum or complement of their inputs.

Operation:	AND (product) of two inputs	OR (sum) of two inputs	NOT (complement) on one input
Expression:	xy , or $x \bullet y$	$x + y$	x'
Logic gate:			

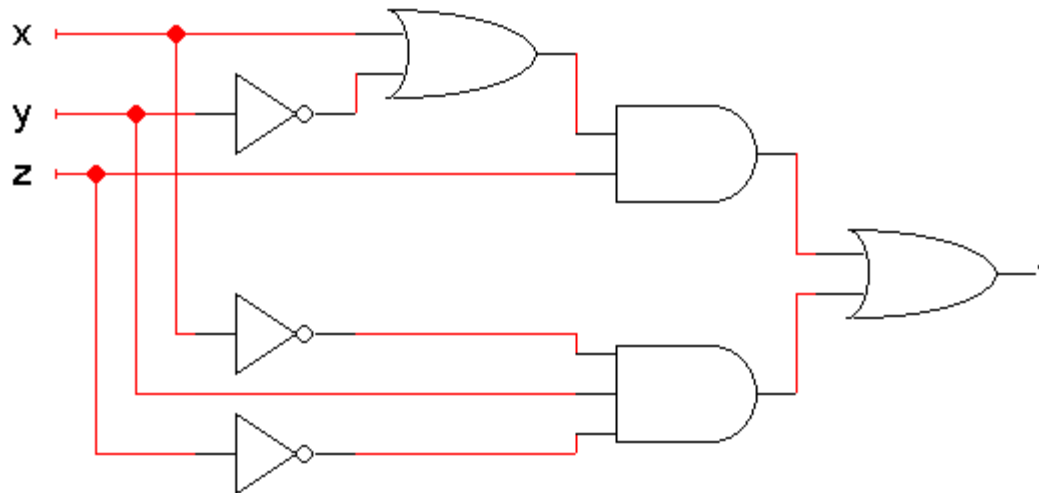
Expressions and circuits

- Any Boolean expression can be converted into a **circuit** by combining basic gates in a relatively straightforward way.
- The diagram below shows the inputs and outputs of each gate.
- The precedence are explicit in a circuit. Clearly, we have to make sure that the hardware does operations in the right order!



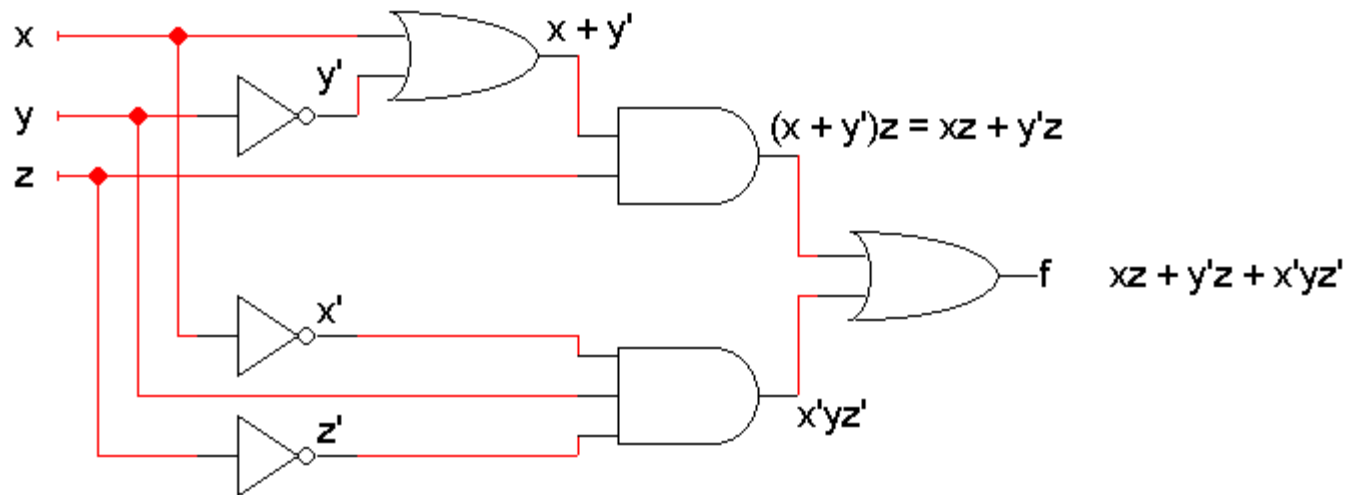
Circuit analysis

- **Circuit analysis** involves figuring out what some circuit does.
 - Every circuit computes some function, which can be described with Boolean expressions or truth tables.
 - So, the goal is to find an expression or truth table for the circuit.
- The first thing to do is figure out what the inputs and outputs of the overall circuit are.
 - This step is often overlooked!
 - The example circuit here has *three* inputs x , y , z and *one* output f .



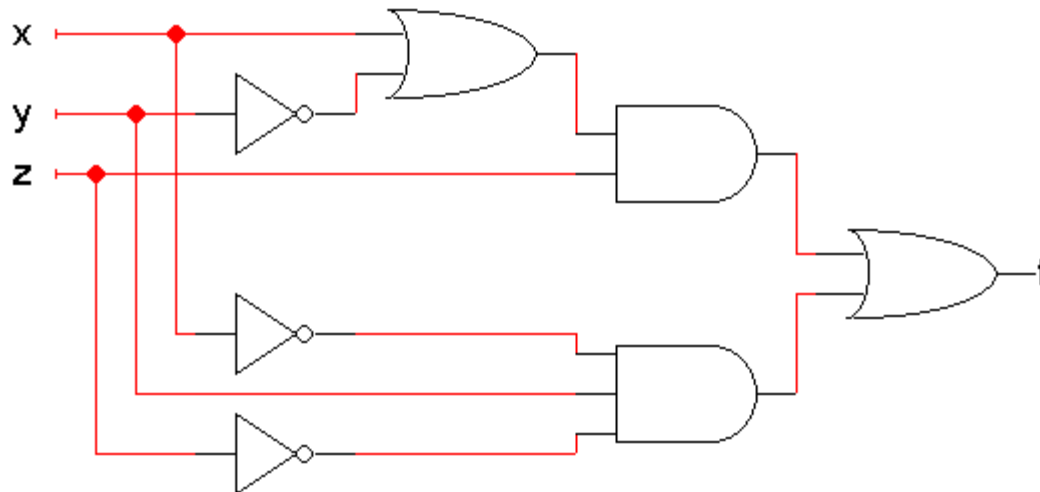
Write algebraic expressions...

- Next, write expressions for the outputs of each individual gate, based on that gate's inputs.
 - Start from the inputs and work towards the outputs.
 - It might help to do some algebraic simplification along the way.
- Here is the example again.
 - We did a little simplification for the top AND gate.
 - You can see the circuit computes $f(x,y,z) = xz + y'z + x'yz'$



...or make a truth table

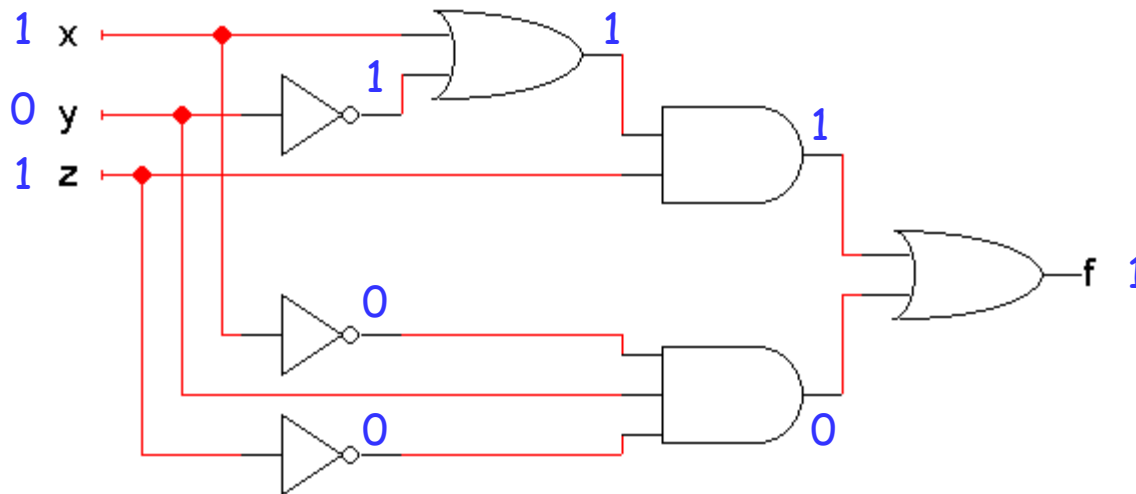
- It's also possible to find a truth table directly from the circuit.
- Once you know the number of inputs and outputs, list all the possible input combinations in your truth table.
 - A circuit with n inputs should have a truth table with 2^n rows.
 - Our example has three inputs, so the truth table will have $2^3 = 8$ rows. All the possible input combinations are shown.



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

Simulating the circuit

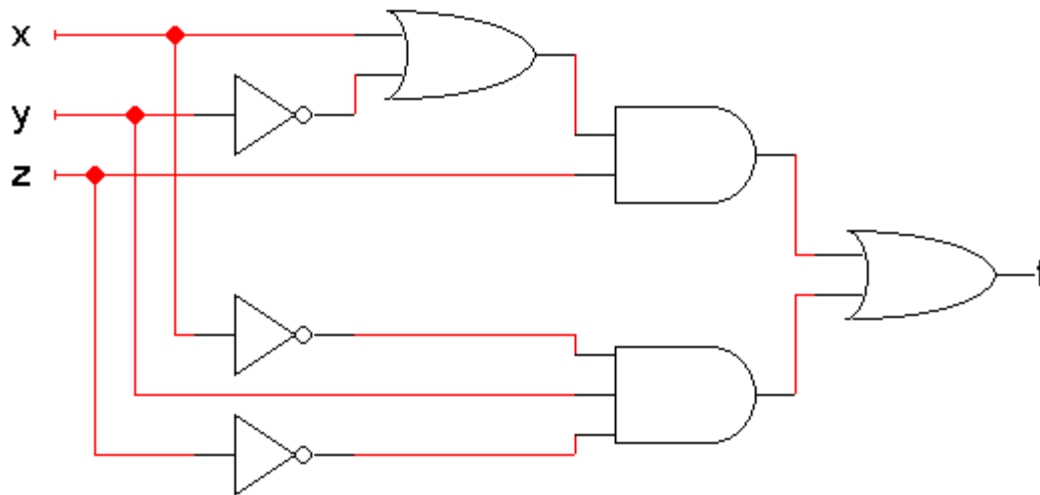
- Then you can simulate the circuit, either by hand or with a program, to find the output for each possible combination of inputs.
- For example, when $xyz = 101$, the gate outputs would be as shown below.
 - Use truth tables for AND, OR and NOT to find the gate outputs.
 - For the final output, we find that $f(1,0,1) = 1$.



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

Finishing the truth table

- Doing the same thing for all the other input combinations yields the complete truth table.
- This is simple, but tedious.



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

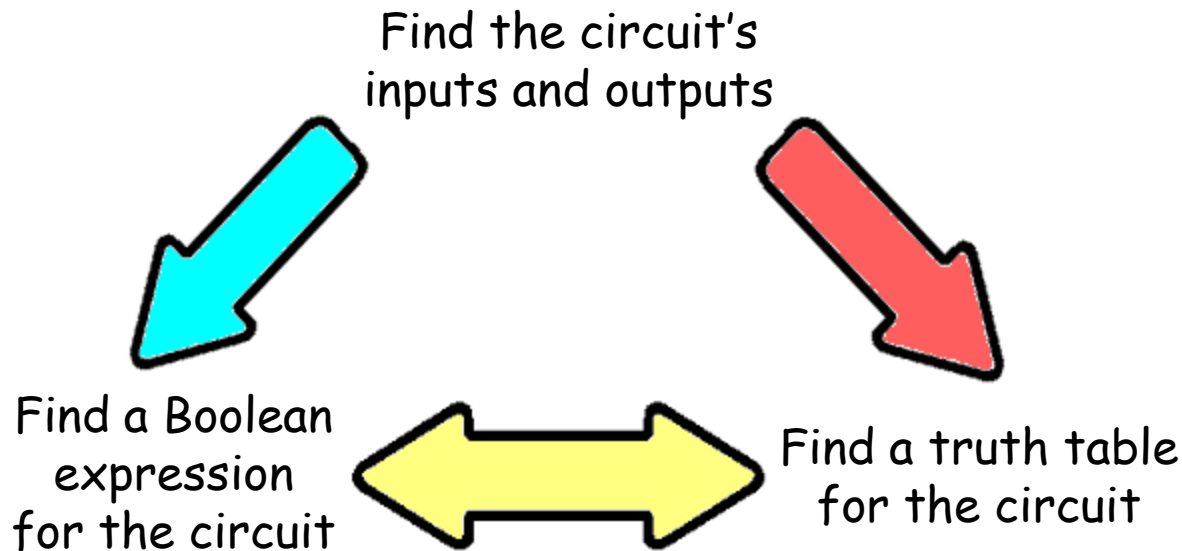
Expressions and truth tables*

- Remember that if you already have a Boolean expression, you can use that to easily make a truth table.
- For example, since we already found that the circuit computes the function $f(x,y,z) = xz + y'z + x'yz'$, we can use that to fill in a table:
 - We show intermediate columns for the terms xz , $y'z$ and $x'yz'$.
 - Then, f is obtained by just OR'ing the intermediate columns.

x	y	z	xz	y'z	x'yz'	f
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	0	1	1
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	1	1	0	0	1

Circuit analysis summary

- After finding the circuit inputs and outputs, you can come up with either an expression or a truth table to describe what the circuit does.
- You can easily convert between expressions and truth tables.



Boolean operations summary

- A variable whose value can be either 1 or 0 is called a Boolean variable.
- AND, OR, and NOT are the basic Boolean operations.
- We can express Boolean functions with either an expression or a truth table.
- Every Boolean expression can be converted to a circuit.
- Now, we'll look at how Boolean algebra can help simplify expressions, which in turn will lead to simpler circuits.

Expression simplification

- Normal mathematical expressions can be simplified using the laws of algebra
- For binary systems, we can use **Boolean algebra**, which is superficially similar to regular algebra
- There are many differences, due to
 - having only two values (0 and 1) to work with
 - having a complement operation
 - the OR operation is not the same as addition

Formal definition of Boolean algebra*

- A Boolean algebra requires
 - A set of elements **B**, which needs *at least* two elements (0 and 1)
 - Two binary (two-argument) operations OR and AND
 - A unary (one-argument) operation NOT
 - The **axioms** below must always be true
 - The **pink axioms** deal with the complement operation
 - **Blue axioms** (especially 15) are different from regular algebra

$$1. x + 0 = x$$

$$2. x \bullet 1 = x$$

$$3. x + 1 = 1$$

$$4. x \bullet 0 = 0$$

$$5. x + x = x$$

$$6. x \bullet x = x$$

$$7. x + x' = 1$$

$$8. x \bullet x' = 0$$

$$9. (x')' = x$$

$$10. x + y = y + x$$

$$11. xy = yx$$

Commutative

$$12. x + (y + z) = (x + y) + z$$

$$13. x(yz) = (xy)z$$

Associative

$$14. x(y + z) = xy + xz$$

$$15. x + yz = (x + y)(x + z)$$

Distributive

$$16. (x + y)' = x'y'$$

$$17. (xy)' = x' + y'$$

DeMorgan's

Are these axioms for real?

- We can show that these axioms are true, given the definitions of AND, OR and NOT

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

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

x	x'
0	1
1	0

- The first 11 axioms are easy to see from these truth tables alone. For example, $x + x' = 1$ because of the middle two lines below (where $y = x'$)

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

Proving the rest of the axioms

- We can make up truth tables to prove (both parts of) DeMorgan's law
- For $(x + y)' = x'y'$ we can make truth tables for $(x + y)'$ and for $x'y'$

x	y	$x + y$	$(x + y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

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

- In each table, the columns on the left (x and y) are the inputs. The columns on the right are outputs.
- In this case, we only care about the columns in blue. The other “outputs” are just to help us find the blue columns.
- Since both of the columns in blue are the same, this shows that $(x + y)'$ and $x'y'$ are equivalent

Simplification with axioms

- We can now start doing some simplifications

$$\begin{aligned}
 & x'y' + xyz + x'y \\
 &= x'(y' + y) + xyz \quad [\text{Distributive; } x'y' + x'y = x'(y' + y)] \\
 &= x' \bullet 1 + xyz \quad [\text{Axiom 7; } y' + y = 1] \\
 &= x' + xyz \quad [\text{Axiom 2; } x' \bullet 1 = x'] \\
 &= (x' + x)(x' + yz) \quad [\text{Distributive}] \\
 &= 1 \bullet (x' + yz) \quad [\text{Axiom 7; } x' + x = 1] \\
 &= x' + yz \quad [\text{Axiom 2}]
 \end{aligned}$$

$$1. \quad x + 0 = x$$

$$3. \quad x + 1 = 1$$

$$5. \quad x + x = x$$

$$7. \quad x + x' = 1$$

$$9. \quad (x')' = x$$

$$2. \quad x \bullet 1 = x$$

$$4. \quad x \bullet 0 = 0$$

$$6. \quad x \bullet x = x$$

$$8. \quad x \bullet x' = 0$$

$$10. \quad x + y = y + x$$

$$12. \quad x + (y + z) = (x + y) + z$$

$$14. \quad x(y + z) = xy + xz$$

$$16. \quad (x + y)' = x'y'$$

$$11. \quad xy = yx$$

$$13. \quad x(yz) = (xy)z$$

$$15. \quad x + yz = (x + y)(x + z)$$

$$17. \quad (xy)' = x' + y'$$

Commutative

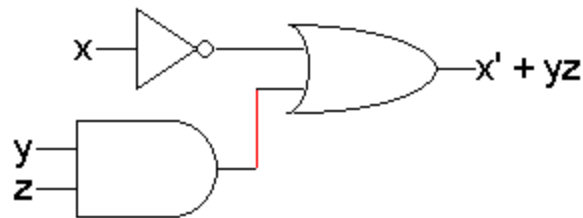
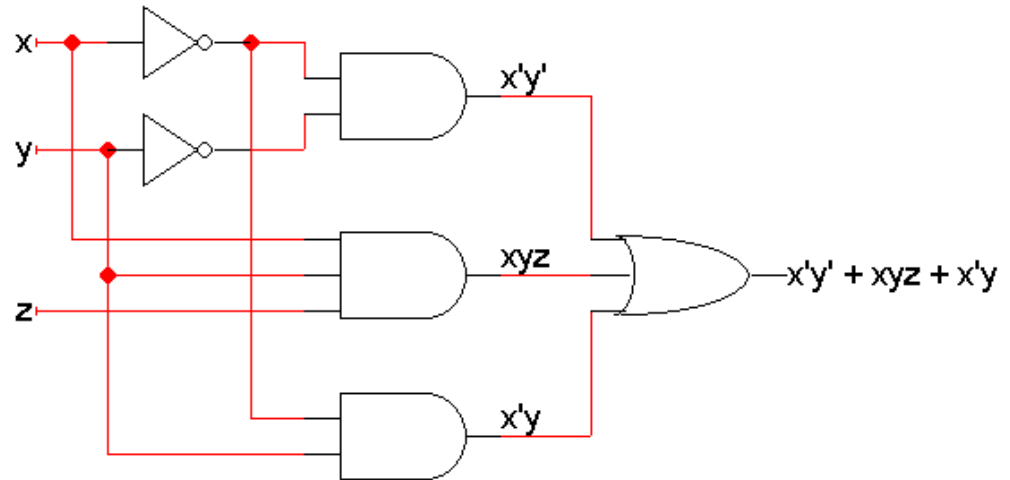
Associative

Distributive

DeMorgan's

Let's compare the resulting circuits

- Here are two different but *equivalent* circuits.
- In general the one with fewer gates is “better”:
 - It costs less to build
 - It requires less power
 - But we had to do some work to find the second form

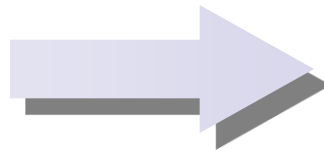


The complement of a function

- The complement of a function always outputs 0 where the original function outputted 1, and 1 where the original produced 0.
- In a truth table, we can just exchange 0s and 1s in the output column(s)

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

x	y	z	$f(x,y,z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



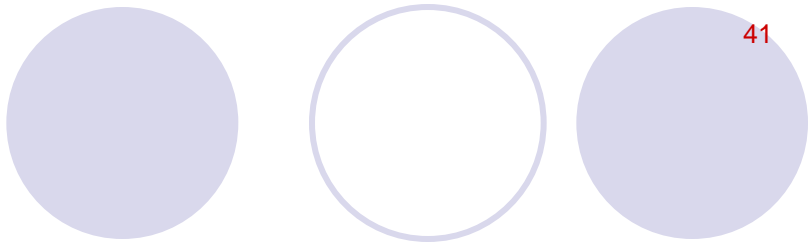
x	y	z	$f'(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Exercise 1

- $F(x_1, x_2, x_3) = (x_1 * x_2) + (\sim x_1 * x_3)$
- What is the truth table? Logic circuit?

Truth Table

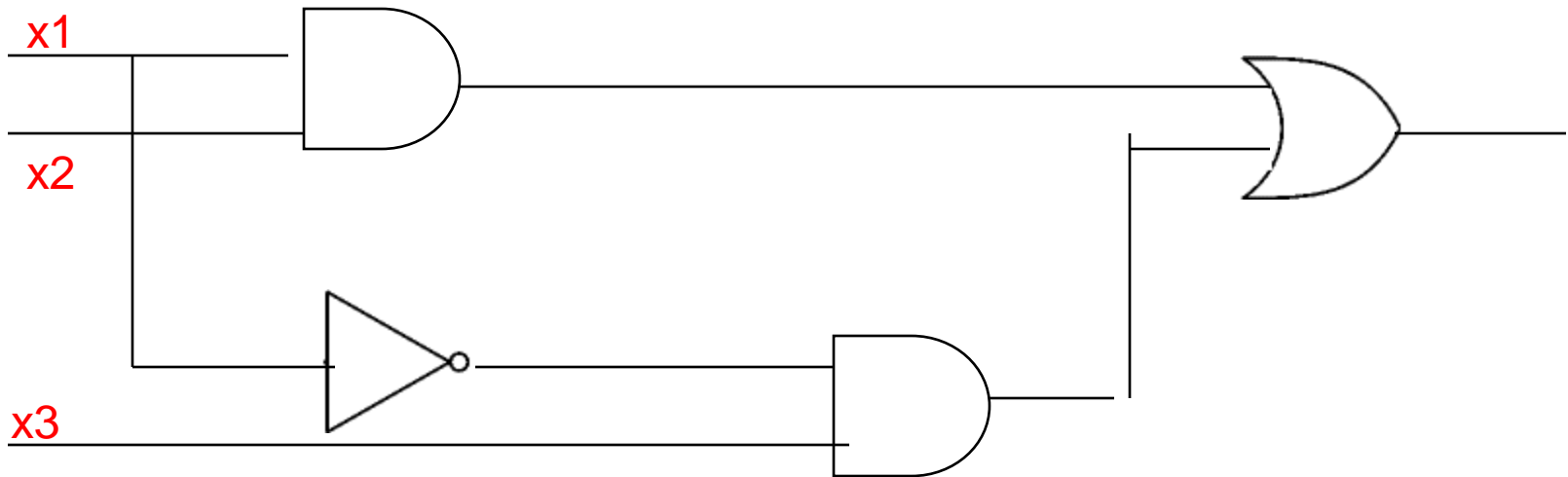
x1	x2	x3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1


$$f(x1, x2, x3) = (x1 * x2) + (\sim x1 * x3)$$

Logic Circuit

42

$$f(x_1, x_2, x_3) = (x_1 * x_2) + (\sim x_1 * x_3)$$



Exercise 2

- $F: \{0, 1\}^3 \rightarrow \{0, 1\}^2$

$$F1 = x + y'.z$$

$$F2 = x'.y'.z + x'.y.z + x.y'$$

- F2 can be simplified as

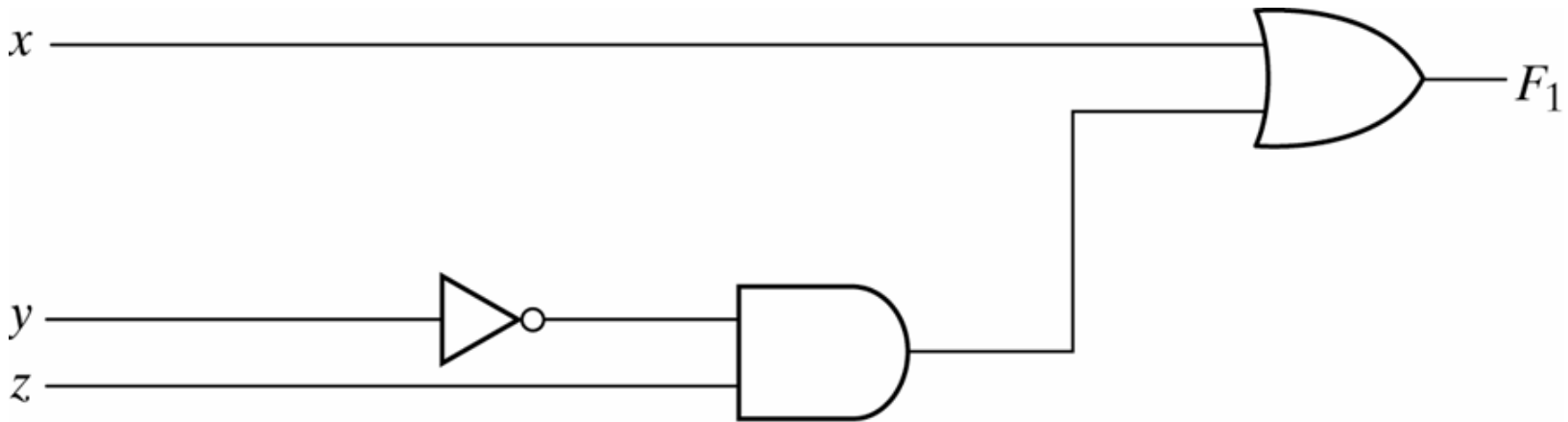
$$F2 = x'.z(y' + y) + x.y' = x'.z + x.y'$$

Exercise 2 -- Truth table

x	y	z	F1	F2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

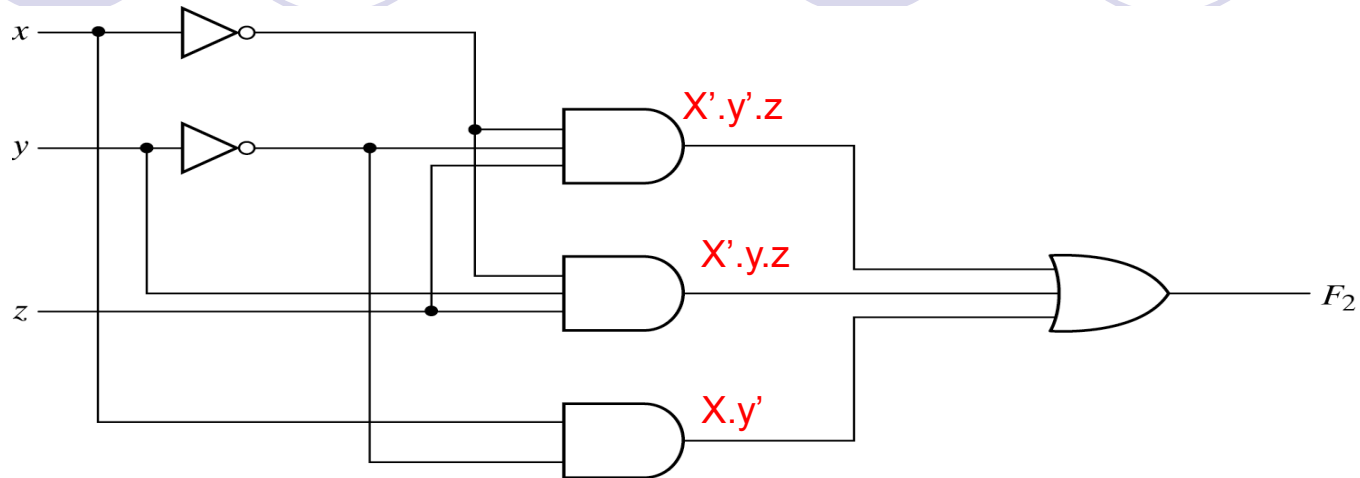
A Boolean Function can be represented in only one truth table forms!

Exercise 2 – logical gate for F_1

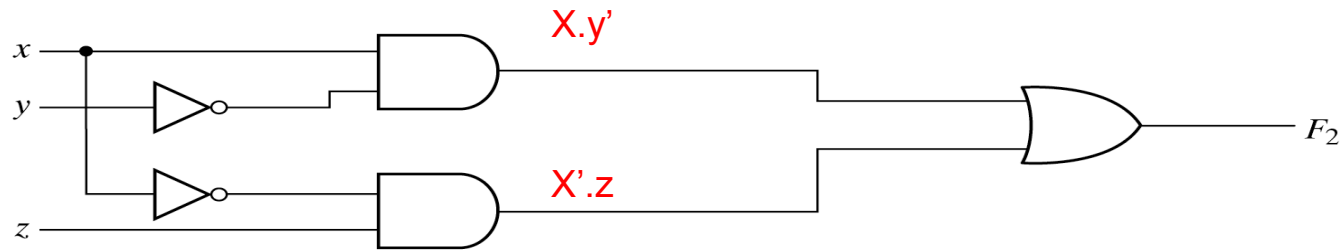


Gate implementation of $F_1 = x + y'z$

Exercise 2 – logical gate for F_2



(a) $F_2 = x'y'z + x'yz + xy'$

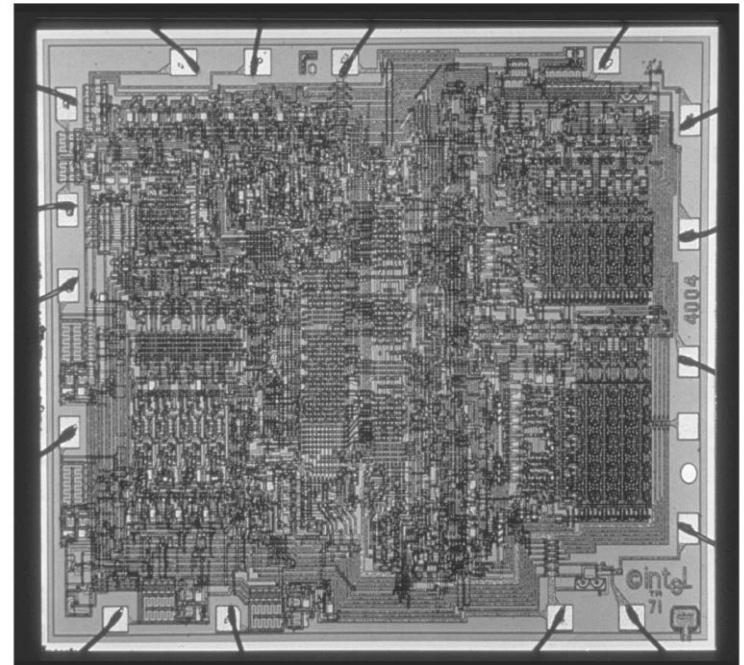


(b) $F_2 = xy' + x'z$

Fig. 2-2 Implementation of Boolean function F_2 with gates

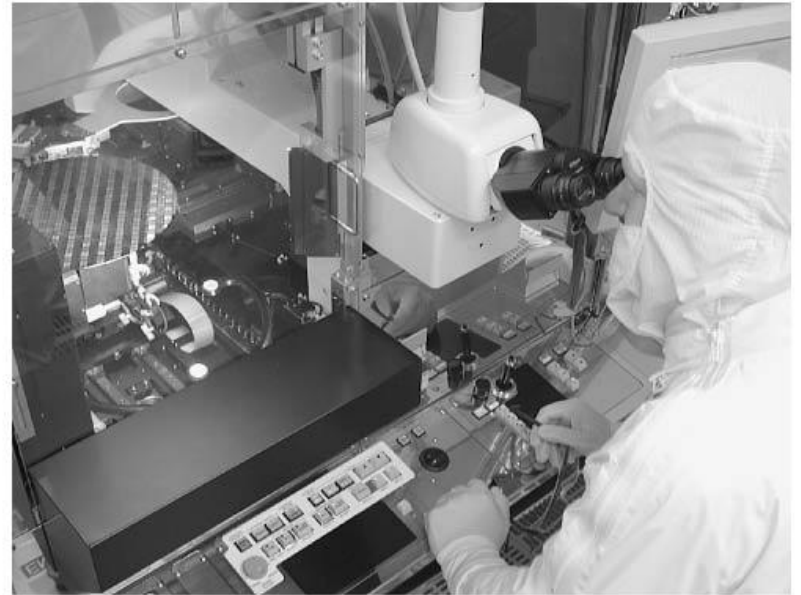
From Circuits to Microchips

- initially, circuits were built by wiring together individual transistors
 - this did not lend itself to mass production
 - it also meant that even simple circuits consisting of tens or hundreds of transistors were quite large (to allow space for human hands)
- in 1958, two researchers (Jack Kilby and Robert Noyce) independently developed techniques that allowed for the mass-production of circuitry
 - circuitry (transistors + connections) is layered onto a single wafer of silicon, known as a *microchip*
 - since every component is integrated onto the same microchip, these circuits became known as *integrated circuits*



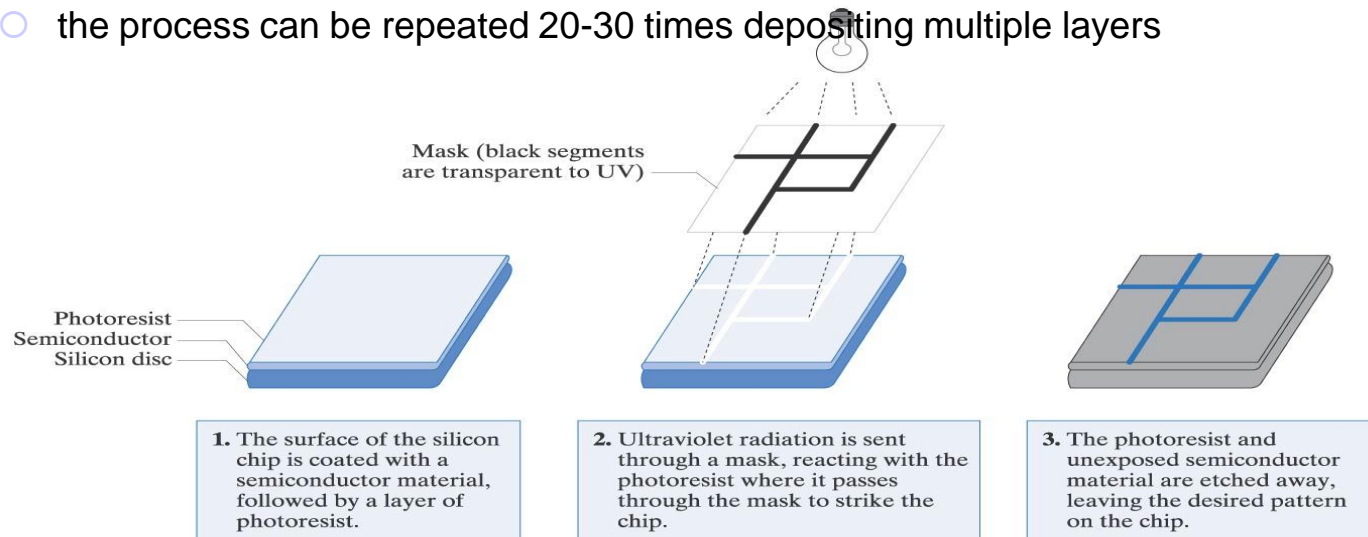
Manufacturing ICs

- the production of integrated circuits is one of the most complex engineering processes in the world



Manufacturing ICs

- to produce the incredibly small and precise circuitry on microchips, manufacturers use light-sensitive chemicals
 - initially, the silicon chip is covered with a semiconductor material, then coated with a layer of photoresist (a chemical sensitive to UV light)
 - transistors are then printed onto a mask (transparent surface on which an opaque coating has been applied to form patterns)
 - UV light is filtered through the mask, passing through the transparent portions and striking the surface of the chip in the specified pattern
 - the photoresist that is exposed to the UV light reacts, hardening the layer of the semiconductor below it
 - the photoresist that was not exposed and the soft layer of semiconductor below are etched away, leaving only the desired pattern of semiconductor material on the surface of the chip
 - the process can be repeated 20-30 times depositing multiple layers



Packaging Microchips

50

- since a silicon chip is fragile, the chip is encased in plastic for protection
 - metal pins are inserted on both sides of the packaging, facilitating easy connections to other microchips
- impact of the microchip
 - lower cost due to mass production
 - faster operation speed due to the close proximity of circuits on chips
 - simpler design/construction of computers using prepackaged components

● Moore's Law describes the remarkable evolution of manufacturing technology

- Moore noted that the number of transistors that can fit on a microchip doubles every 12 to 18 months
- this pattern has held true for the past 30 years
- industry analysts predict that it will continue to hold for the near future

