Boolean Algebra and Digital Logic

Hamza Osman İLHAN

hoilhan@yildiz.edu.tr

D-037

Objectives

- Understand the relationship between Boolean logic and digital computer circuits.
- Learn how to design simple logic circuits.
- Understand how digital circuits work together to form complex computer systems.

Introduction

- In the latter part of the nineteenth century, George Boole incensed philosophers and mathematicians alike when he suggested that logical thought could be represented through mathematical equations.
  - *How dare anyone suggest that human thought could be encapsulated and manipulated like an algebraic formula?*
- Computers, as we know them today, are implementations of Boole's *Laws of Thought*.
  - John Atanasoff and Claude Shannon were among the first to see this connection.

- In the middle of the twentieth century, computers were commonly known as “thinking machines” and “electronic brains.”
  - Many people were fearful of them.
- Nowadays, we rarely ponder the relationship between electronic digital computers and human logic. Computers are accepted as part of our lives.
  - Many people, however, are still fearful of them.
- In this chapter, you will learn the simplicity that constitutes the essence of the machine.

Boolean Algebra

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.
  - In formal logic, these values are "true" and "false."
  - In digital systems, these values are "on" and "off," 1 and 0, or "high" and "low."
- Boolean expressions are created by performing operations on Boolean variables.
  - Common Boolean operators include AND, OR, and NOT.

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

X AND Y

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

X OR Y

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

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

NOT x

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

- 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}.

- The truth table for the Boolean function:

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

is shown at the right.

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

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

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

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

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

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

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

- Most Boolean identities have an AND (product) form as well as an OR (sum) form. We give our identities using both forms. Our first group 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	$x\overline{x} = 0$	$x + \overline{x} = 1$

- Our 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	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$

- Our last 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	$\overline{(xy)} = \overline{x} + \overline{y}$	$\overline{(x+y)} = \overline{x}\,\overline{y}$
Double Complement Law	$\overline{(\overline{x})} = x$	

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

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

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

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

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

is:

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

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

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

$$= (\bar{X}+\bar{Y})(X+\bar{Z})(\bar{Y}+Z)$$

- Through our exercises in simplifying Boolean expressions, we see that there are numerous ways of stating the same Boolean expression.
  - These “synonymous” forms are *logically equivalent.*
  - Logically equivalent expressions have identical truth tables.
- In order to eliminate as much confusion as possible, designers express Boolean functions in *standardized* or *canonical* form.

- There are two canonical forms for Boolean expressions: sum-of-products and product-of-sums.
  - Recall 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-s … together:
  - For example:

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

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

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

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

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

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

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

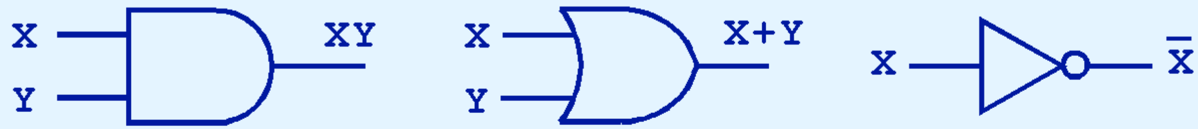
$$F(x,y,z) = x\bar{z}+y$$

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

Logic Gates

- We have looked at Boolean functions in abstract terms.
- In this section, we see that Boolean functions are implemented in digital computer circuits called gates.
- A gate is an electronic device that produces a result based on two or more 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.

- 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

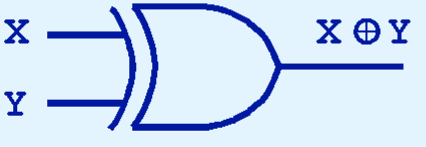
X	$\overline{X}$
0	1
1	0

- They corr ... ons, as you ca

- 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

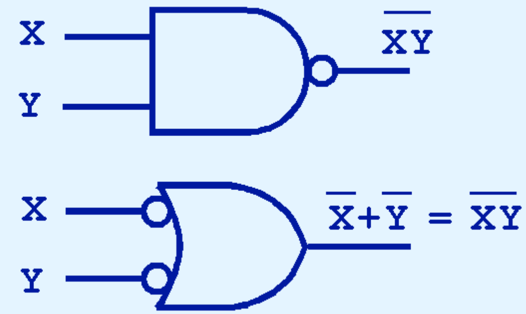


Note the special symbol ⊕ for the XOR operation.

- NAND and NOR are two very important gates. Their symbols and truth tables are shown at the right.

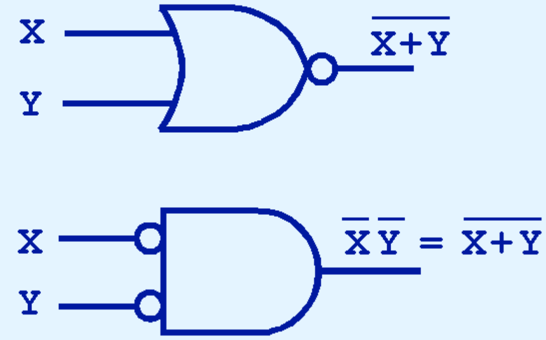
X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

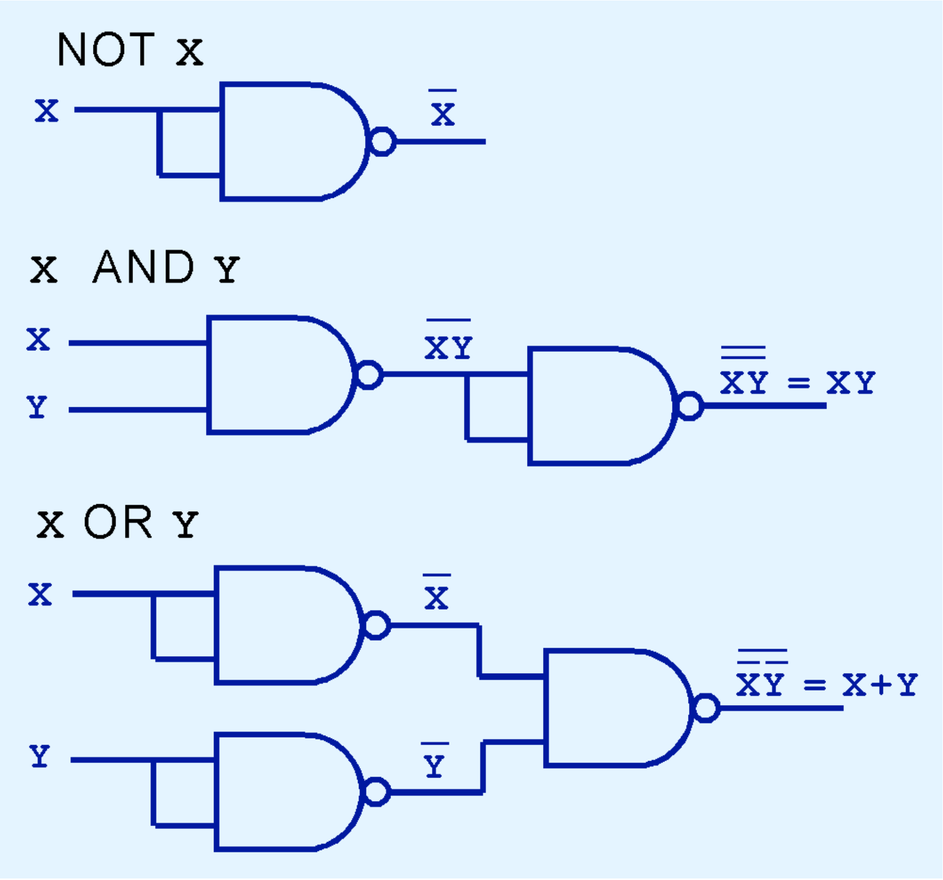


X NOR Y

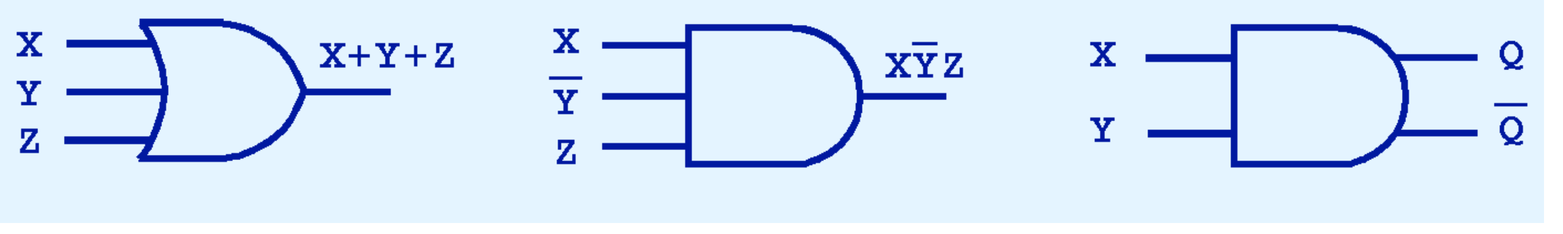
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



- NAND and NOR are known as *universal gates* because they are inexpensive to manufacture and any Boolean function can be constructed using only NAND or only NOR gates.



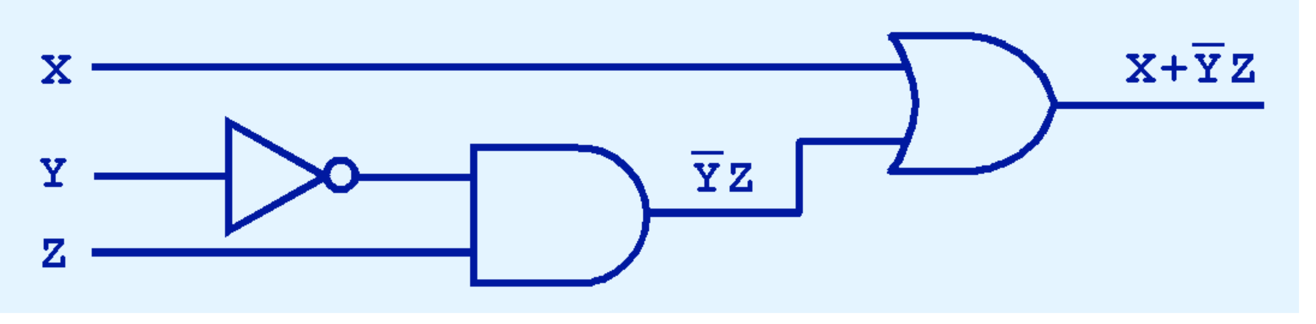
- Gates can have multiple inputs and more than one output.
  - A second output can be provided for the complement of the operation.
  - We'll see more of this later.



Digital Components

- The main thing to remember is that combinations of gates implement Boolean functions.
- The circuit below implements the Boolean function:

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



We simplify our Boolean expressions so that we can create simpler circuits.

- We have designed a circuit that implements the Boolean function:

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

- This circuit is an example of a *combinational logic* circuit.
- Combinational logic circuits produce a specified output (almost) at the instant when input values are applied.
  - In a later section, we will explore circuits where this is not the case.

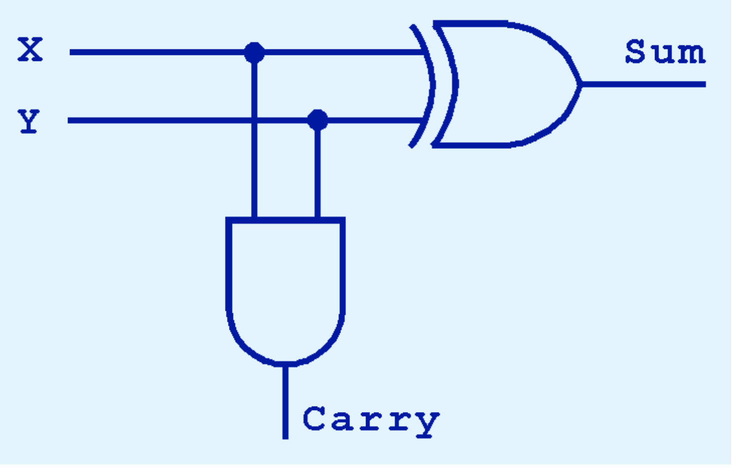
Combinational Circuits

- Combinational logic circuits give us many useful devices.
- One of the simplest is the *half adder*, which finds the sum of two bits.
- We can gain some insight as to the construction of a half adder by looking at its truth table, shown at the right.

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

- As we see, the sum can be found using the XOR operation and the carry using the AND operation.

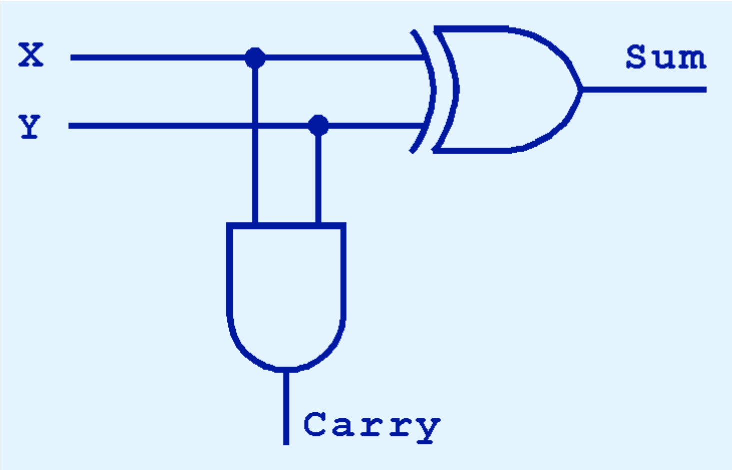
Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



- We can change our half adder into to a full adder by including gates for processing the carry bit.
- The truth table for a full adder is shown at the right.

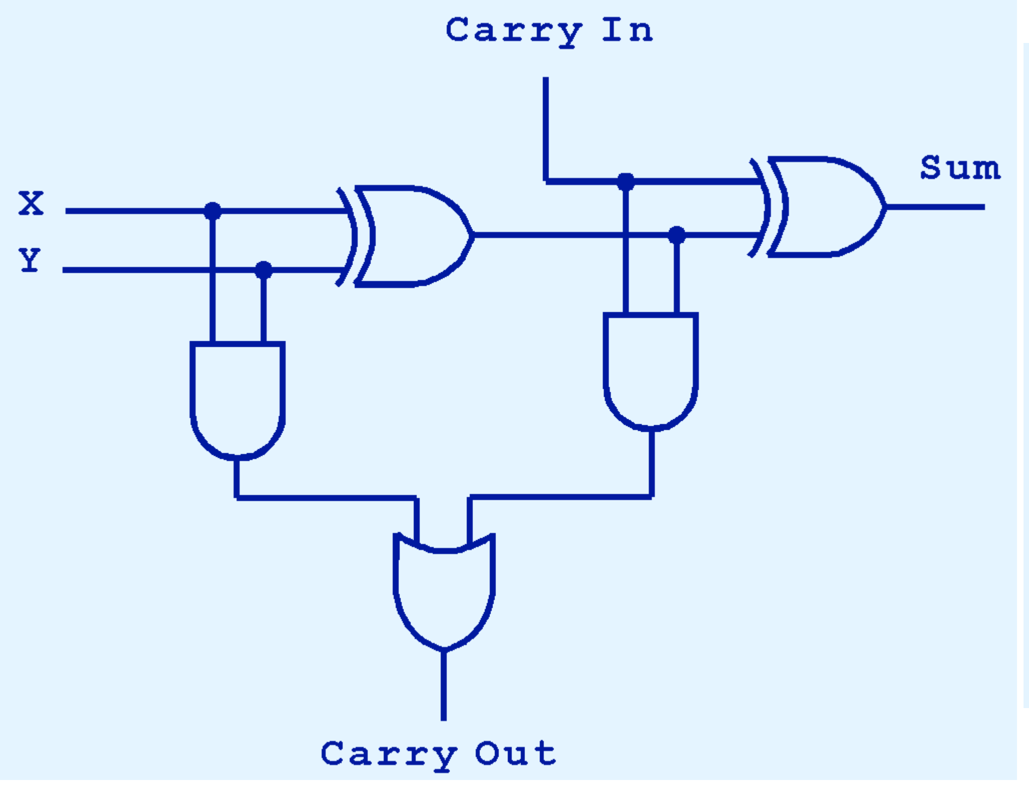
Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- How can we change the half adder shown below to make it a full adder?



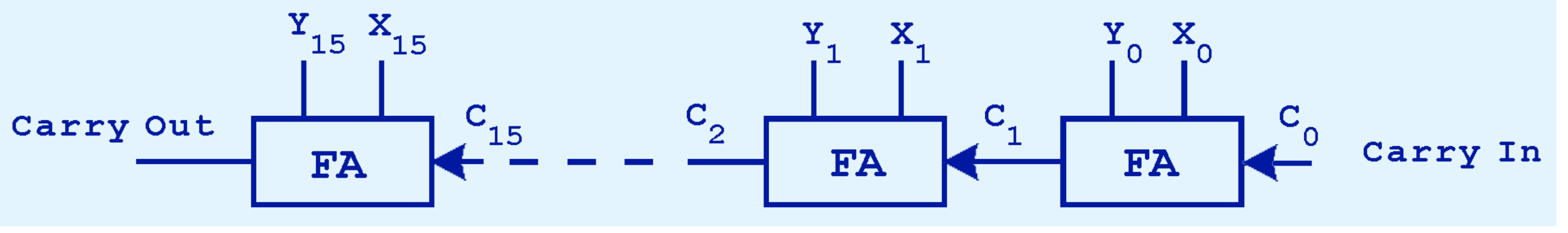
Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Here's our completed full adder.



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

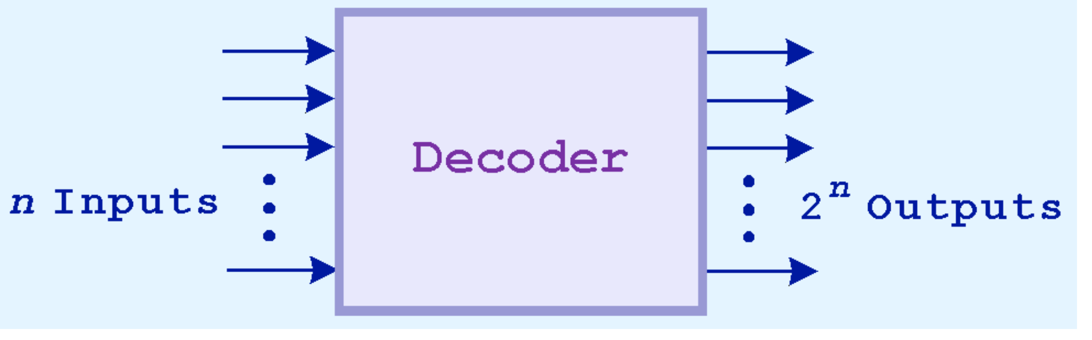
- Just as we combined half adders to make a full adder, full adders can connected in series.
- The carry bit “ripples” from one adder to the next; hence, this configuration is called a *ripple-carry adder*.



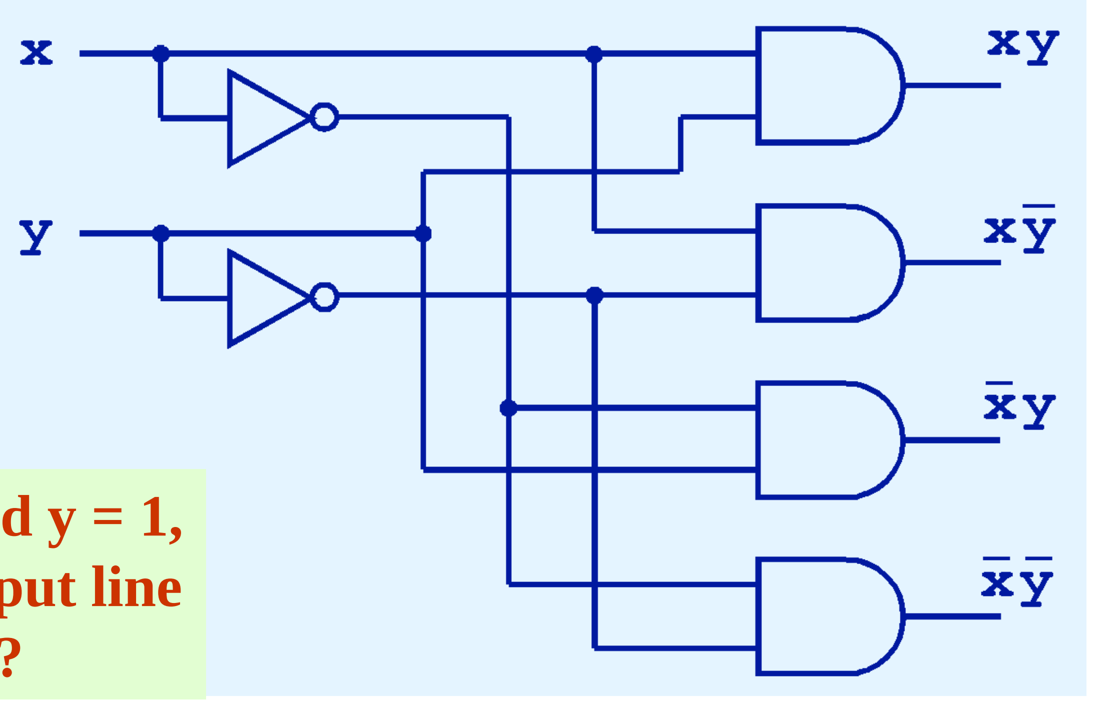
Today’s systems employ more efficient adders.

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

This is a block diagram for a decoder.

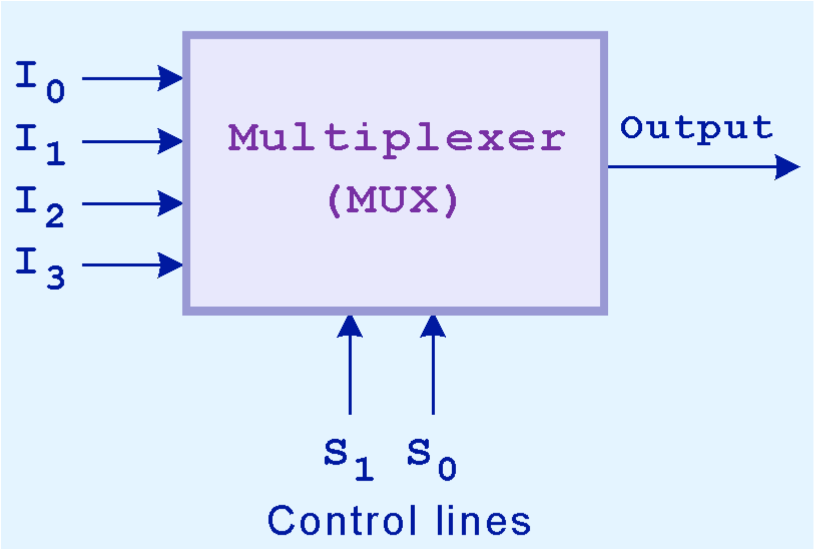


- This is what a 2-to-4 decoder looks like on the inside.



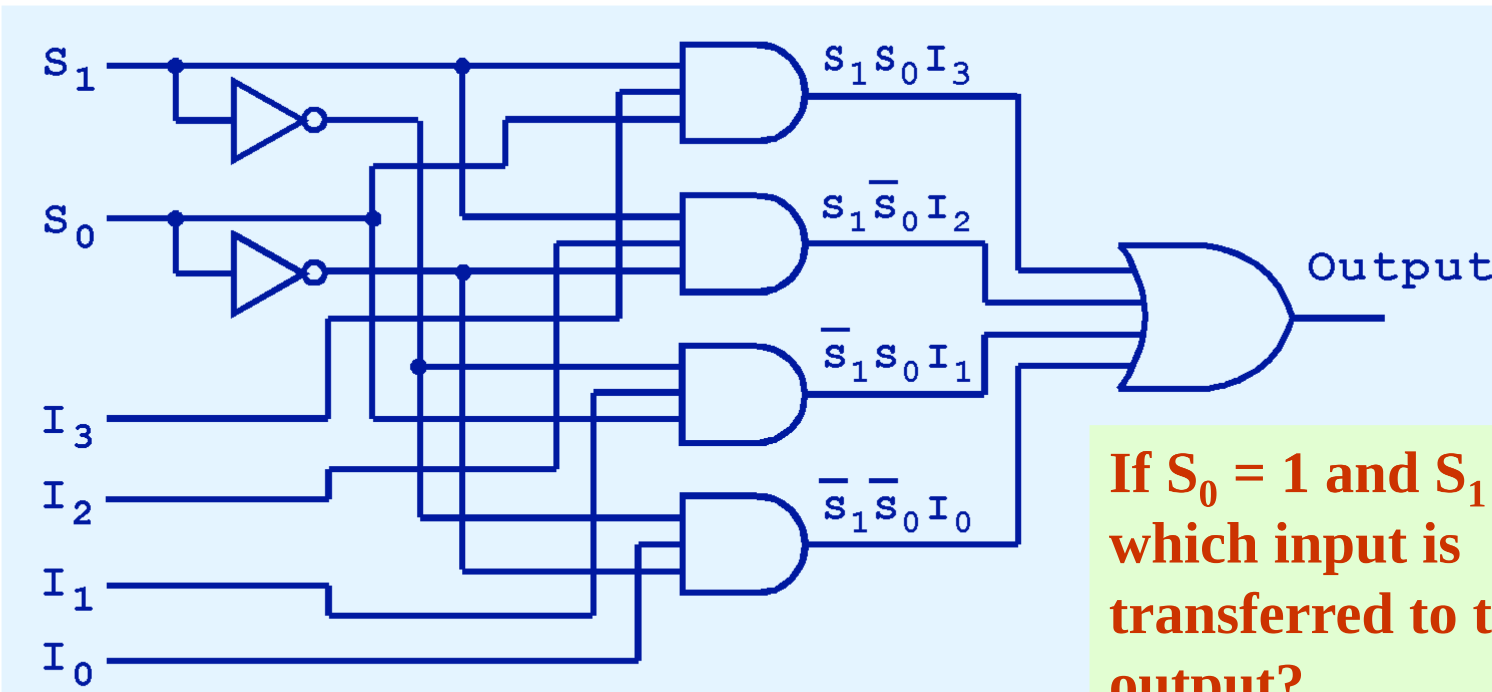
If x = 0 and y = 1, which output line is enabled?

- 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.



This is a block diagram for a multiplexer.

- This is what a 4-to-1 multiplexer looks like on the inside.

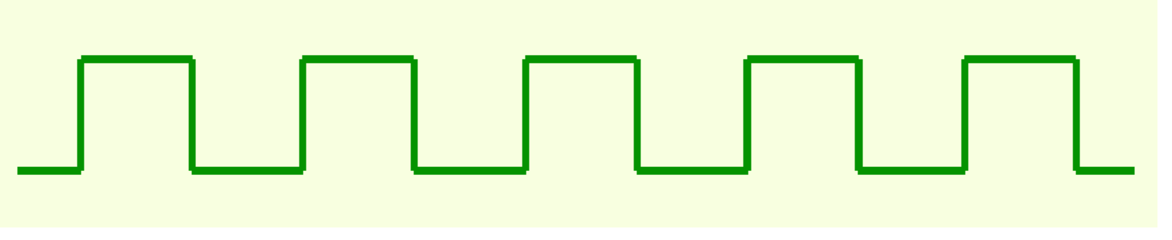


If S_0 = 1 and S_1 = 0, which input is transferred to the output?

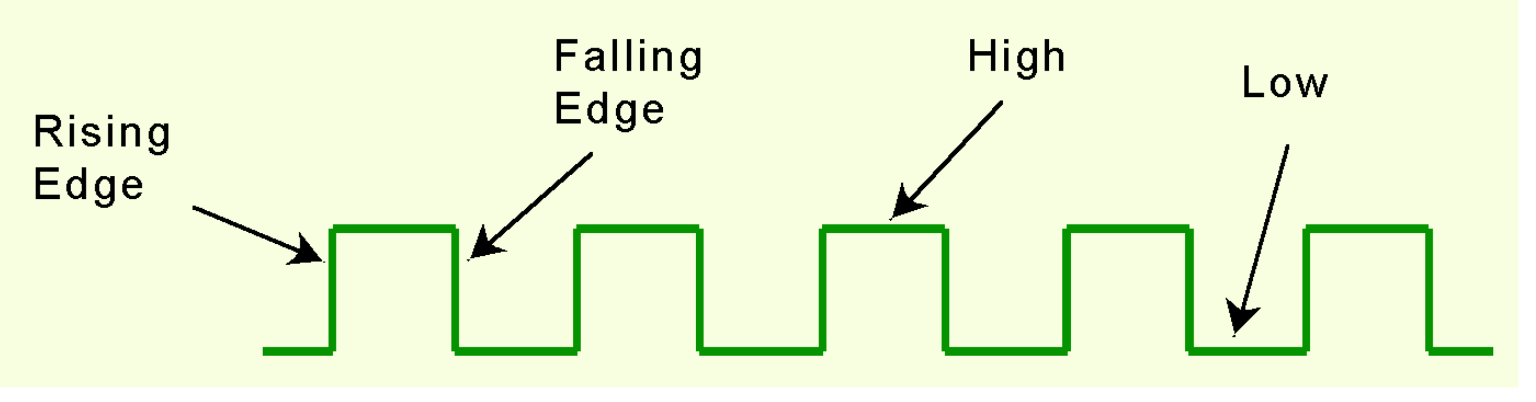
Sequential Circuits

- Combinational logic circuits are perfect for situations when we require the immediate application of a Boolean function to a set of inputs.
- There are other times, however, when we need a circuit to change its value with consideration to its current state as well as its inputs.
  - These circuits have to "remember" their current state.
- *Sequential logic circuits* provide this functionality for us.

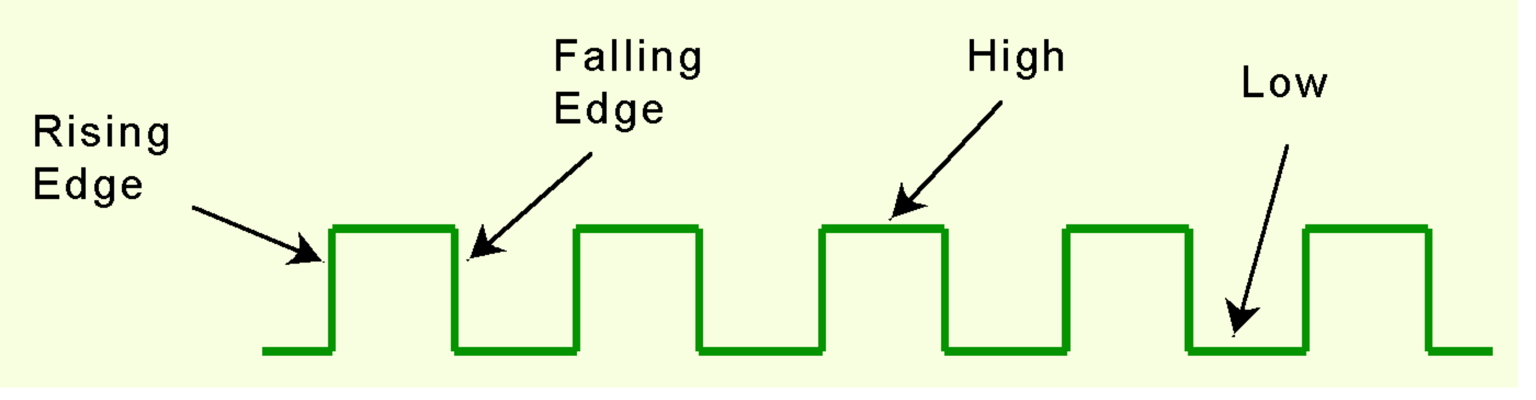
- As the name implies, sequential logic circuits require a means by which events can be sequenced.
- State changes are controlled by clocks.
  - A “clock” is a special circuit that sends electrical pulses through a circuit.
- Clocks produce electrical waveforms such as the one shown below.



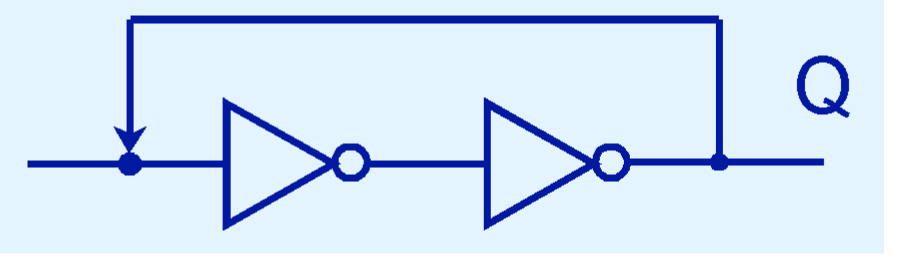
- State changes occur in sequential circuits only when the clock ticks.
- Circuits can change state on the rising edge, falling edge, or when the clock pulse reaches its highest voltage.



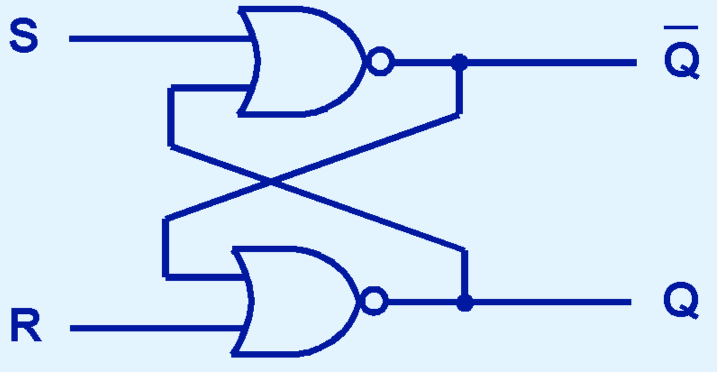
- Circuits that change state on the rising edge, or falling edge of the clock pulse are called *edge-triggered.*
- *Level-triggered circuits* change state when the clock voltage reaches its highest or lowest level.

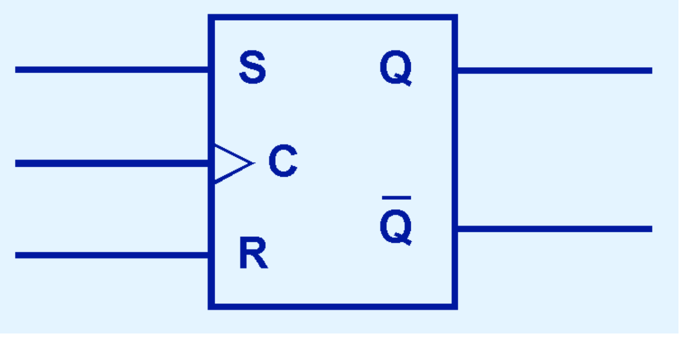


- To retain their state values, sequential circuits rely on *feedback.*
- Feedback in digital circuits occurs when an output is looped back to the input.
- A simple example of this concept is shown below.
  - If Q is 0 it will always be 0, if it is 1, it will always be 1. Why?

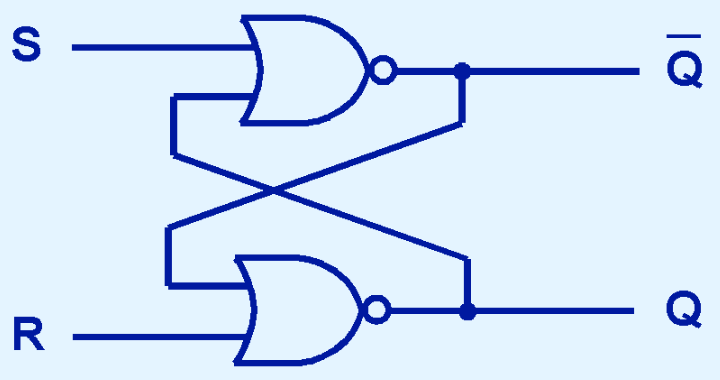


- You can see how feedback works by examining the most basic sequential logic components, the SR flip-flop.
  - The “SR” stands for set/reset.
- The internals of an SR flip-flop are shown below, along with its block diagram.





- The behavior of an SR flip-flop is described by a characteristic table.
- Q(t) means the value of the output at time t. Q(t+1) is the value of Q after the next clock pulse.



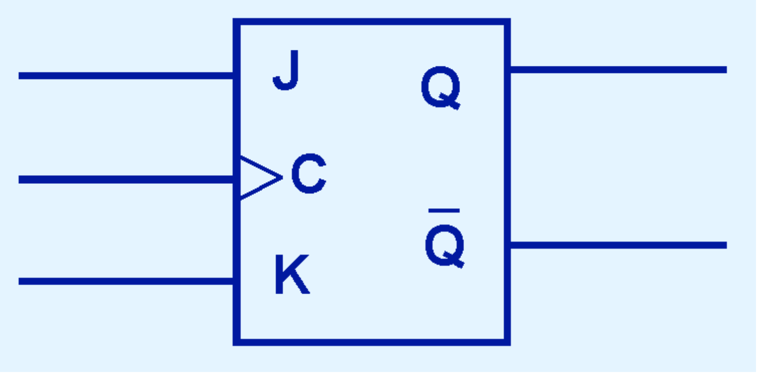
S	R	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

- The SR flip-flop actually has three inputs: S, R, and its current output, Q.
- Thus, we can construct a truth table for this circuit, as shown at the right.
- Notice the two undefined values. When both S and R are 1, the SR flip-flop is unstable.

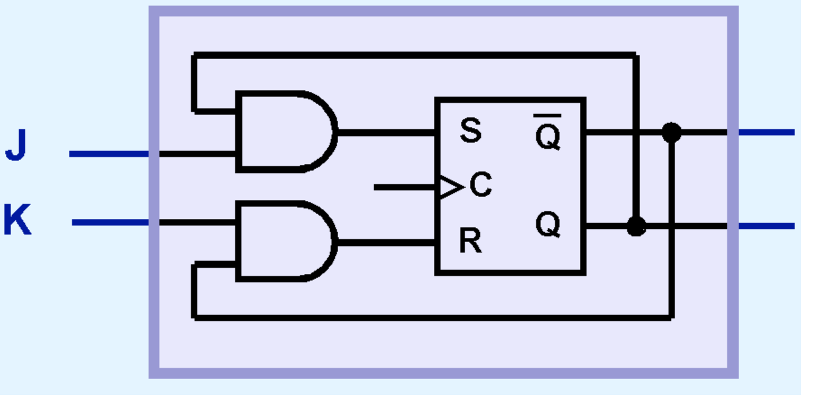
S	R	Present State Q(t)	Next State Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

- If we can be sure that the inputs to an SR flip-flop will never both be 1, we will never have an unstable circuit. This may not always be the case.
- The SR flip-flop can be modified to provide a stable state when both inputs are 1.

- This modified flip-flop is called a JK flip-flop, shown at the right.
  - The "JK" is in honor of Jack Kilby.

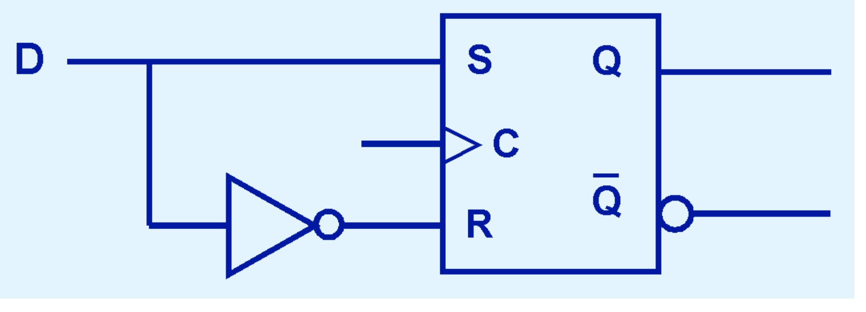


- At the right, we see how an SR flip-flop can be modified to create a JK flip-flop.
- The characteristic table indicates that the flip-flop is stable for all inputs.



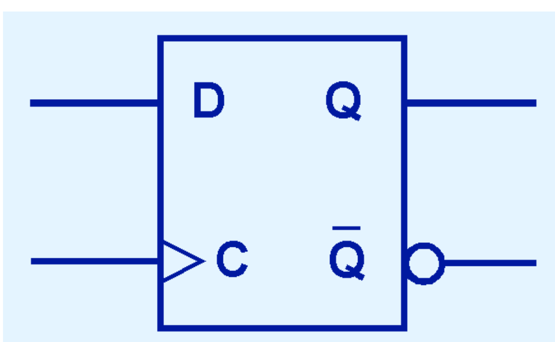
J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\overline{Q}(t)$

- Another modification of the SR flip-flop is the D flip-flop, shown below with its characteristic table.
- You will notice that the output of the flip-flop remains the same during subsequent clock pulses. The output changes only when the value of D changes.



D	Q(t+1)
0	0
1	1

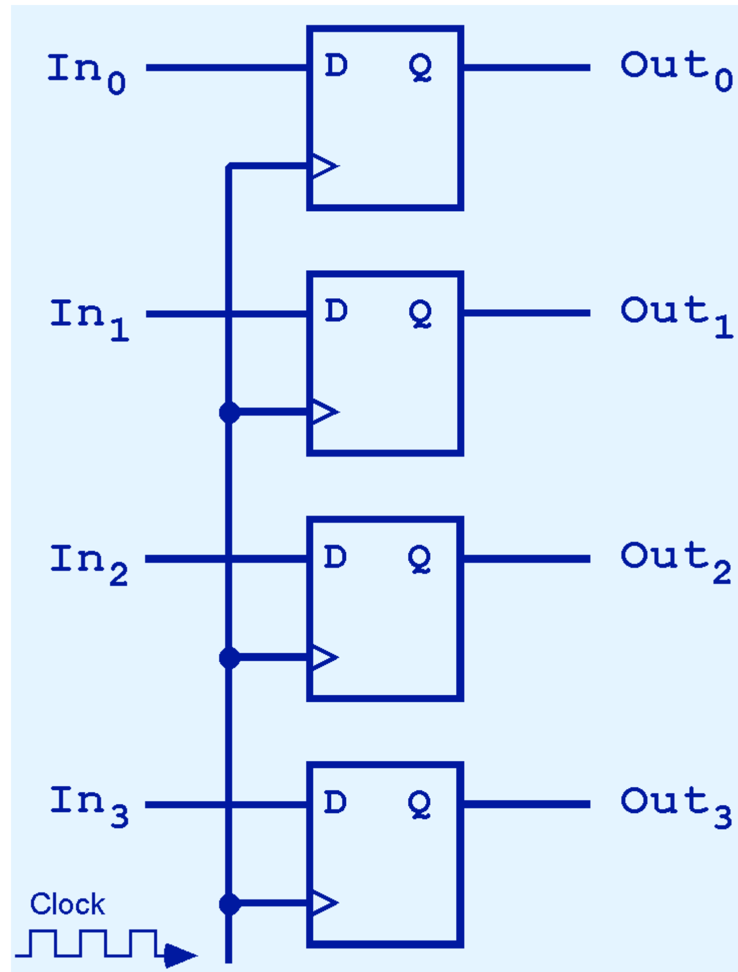
- The D flip-flop is the fundamental circuit of computer memory.
  - D flip-flops are usually illustrated using the block diagram shown below.
- The next slide shows how these circuits are combined to create a register.



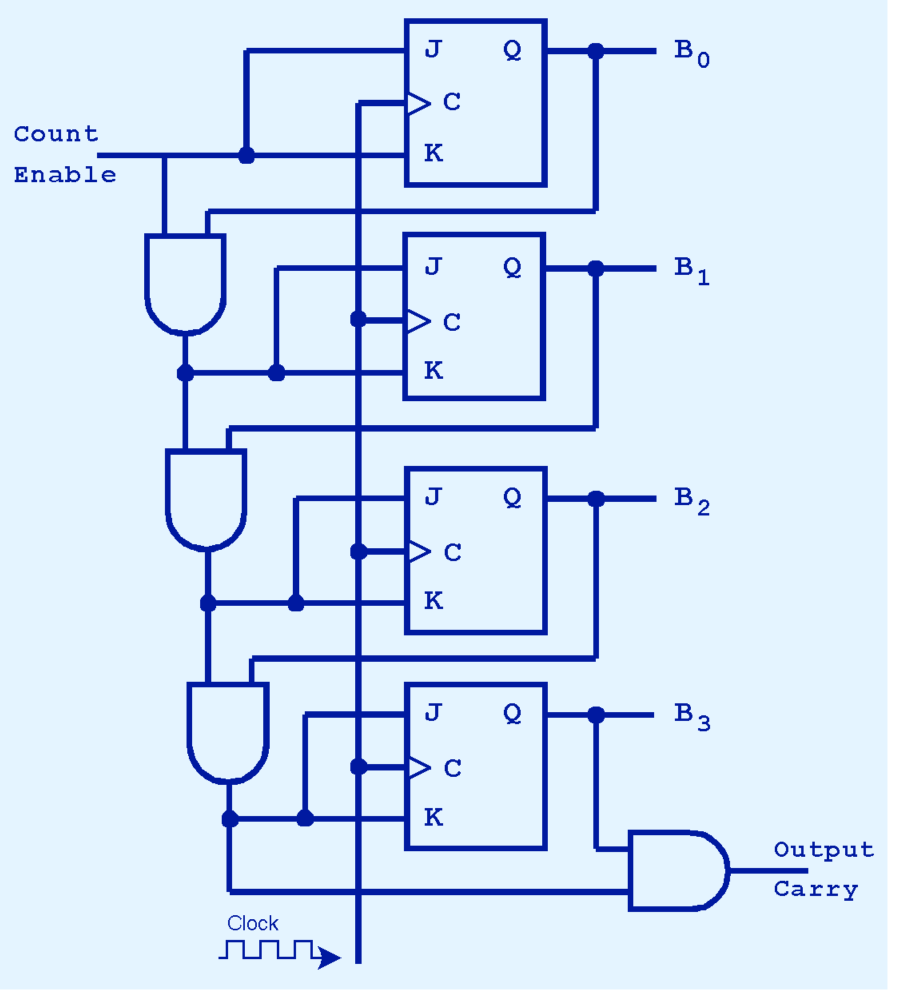
D	Q(t+1)
0	0
1	1

- This illustration shows a 4-bit register consisting of D flip-flops. You will usually see its block diagram (below) instead.

A larger memory configuration is in your text.



- A binary counter is another example of a sequential circuit.
- The low-order bit is complemented at each clock pulse.
- Whenever it changes from 0 to 1, the next bit is complemented, and so on through the other flip-flops.



Designing Circuits

- We have seen digital circuits from two points of view: digital analysis and digital synthesis.
  - *Digital analysis* explores the relationship between a circuits inputs and its outputs.
  - *Digital synthesis* creates logic diagrams using the values specified in a truth table.
- Digital systems designers must also be mindful of the physical behaviors of circuits to include minute propagation delays that occur between the time when a circuit’s inputs are energized and when the output is accurate and stable.

- Digital designers rely on specialized software to create efficient circuits.
  - Thus, software is an enabler for the construction of better hardware.
- Of course, software is in reality a collection of algorithms that could just as well be implemented in hardware.
  - Recall the Principle of Equivalence of Hardware and Software.

- When we need to implement a simple, specialized algorithm and its execution speed must be as fast as possible, a hardware solution is often preferred.
- This is the idea behind *embedded systems*, which are small special-purpose computers that we find in many everyday things.
- Embedded systems require special programming that demands an understanding of the operation of digital circuits, the basics of which you have learned in this chapter.