

## Chapter 3 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.

2

## 3.1 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 the connection between Boole's *Laws of Thought* and the computer circuits.

3

## 3.1 Introduction

- 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.

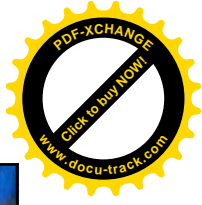
4



## 3.2 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 variables.
  - Common Boolean operators include AND, OR, and NOT.

5



## 3.2 Boolean Algebra

- 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.

X AND Y

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

- The AND operator is also known as a **Boolean product**. The OR operator is the **Boolean sum**.

X OR Y

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

6

## 3.2 Boolean Algebra

- 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” (  $\neg$  ).

NOT X

x	$\bar{x}$
0	1
1	0

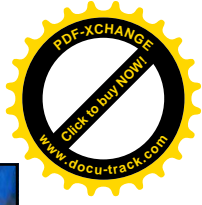
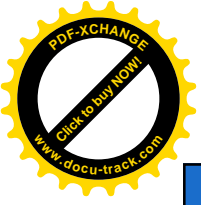
7

## 3.2 Boolean Algebra

- Although there are other Boolean operations, all data calculations and processing inside a computer system are based on and can be performed as a sequence of three Boolean operations (AND, OR, NOT).
- A sequence of Boolean operators with variables yields a Boolean **expression**.  
e.g.  $xy + xz' + y'z$
- A Boolean **function** defines a Boolean expression that maps one or more input values to one output value.

e.g.,  $f(x,y,z) = xy + xz' + y'z$

8



## 3.2 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}.
- A Boolean function can be represented by both a Boolean expression or a truth table. The truth table lists all the possible inputs to the function and the corresponding output values.

Now you know why the binary numbering system is so handy in digital systems.

## 3.2 Boolean Algebra

- The truth table for the Boolean **function**:

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

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

is shown at the right. The expression reads: (x & not z) or 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

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

## 3.2 Boolean Algebra

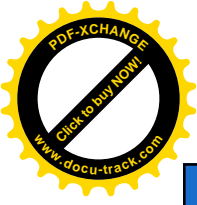
- As with common arithmetic, Boolean operations have rules of precedence.
- The NOT operator has highest priority, followed by AND and then OR (try to remember **NAO** – like **NATO** without the ‘T’).
- This is how we chose the (shaded) function subparts in our table.

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

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

## 3.2 Boolean Algebra

- Frequently, a Boolean function is not in its simplest form, e.g.,  $f(x) = (8x + 5x)$ .
- Digital computers contain circuits (chips) 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, i.e., laws, that help us to do this.



## 3.2 Boolean Algebra

- 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	$x\bar{x} = 0$	$x + \bar{x} = 1$

13



## 3.2 Boolean Algebra

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

**Mnemonics:**

**Commutative – ‘commute’ - something switches positions.**

**Associative - who will you associate with – the value on the right, or the one on the left?**

**Distributive – distribute the value (here ‘x’).**

14

## 3.2 Boolean Algebra

- 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	$x(x+y) = x$	$x + x\bar{y} = x$
DeMorgan's Law	$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x+y)} = \bar{x}\bar{y}$
Double Complement Law	$\overline{(\bar{x})} = x$	

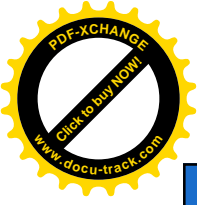
15

## 3.2 Boolean Algebra

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

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

16

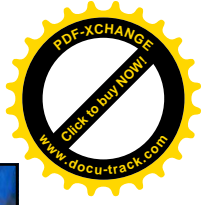


## 3.2 Boolean Algebra

- Quite often, 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.
- If we implement the complement, we must invert the final output to yield the original function; this is accomplished with one simple NOT operation.
- DeMorgan's law provides an easy way of finding the complement of a Boolean function.
- Recall DeMorgan's law states (the long overbar here means 'the complement of'):

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

17



## 3.2 Boolean Algebra

- 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:

$$\begin{aligned}
 F(x, y, z) &= (xy) + (\bar{x}z) + (y\bar{z}) \\
 \text{is: } \overline{F(x, y, z)} &= \overline{(xy) + (\bar{x}z) + (y\bar{z})} \\
 &= \overline{(xy)} \overline{(\bar{x}z)} \overline{(y\bar{z})} \\
 &= (\bar{x} + \bar{y})(x + \bar{z})(\bar{y} + z)
 \end{aligned}$$

You might find it easier to go from the original function directly to the final result by applying DeMorgan's law.

18

## 3.2 Boolean Algebra

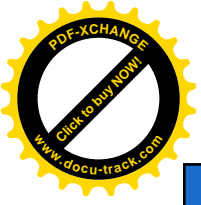
- Through our exercises in simplifying Boolean expressions, we see that there are numerous ways of stating the same Boolean expression.
  - These “synonymous” forms are *logically equivalent*.
  - You can find if two expressions are logically equivalent by:
    1. Creating truth tables for the two expressions and see if they are equal, or
    2. Use the rules of laws (identities) and determine if one expression can be rewritten as another.
- In order to eliminate as much confusion as possible, designers express Boolean functions in *standardized* or *canonical* form.

19

## 3.2 Boolean Algebra

- There are two canonical (**standardized**) 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-sums form, ORed variables are ANDed together:
  - For example:  $F(x, y, z) = (x+y)(x+z)(y+z)$

20

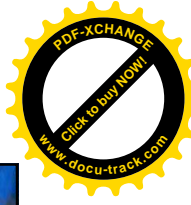


### 3.2 Boolean Algebra

- To find the standard sum-of-products form of the function, first create a truth table for the function.
- We are only interested in the values of the variables that make the function true (=1) – **circuits are designed to 'execute' the function.**
- Using the truth table, we list the values of the variables that result in a true function value:  
010 011 100 110 111  
x'yz' x'yz xy'z' xyz' xyz
- Each group of variables is then ORed together – see on next slide:

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

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



### 3.2 Boolean Algebra

- The sum-of-products form is:  
$$F(x, y, z) = \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z} + xyz$$
- The product-of-sum form can be found by listing all the sum terms that make the function false (=0):  
$$F(x, y, z) = (x+y+z)(x+y+z')(x'+y+z')$$

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

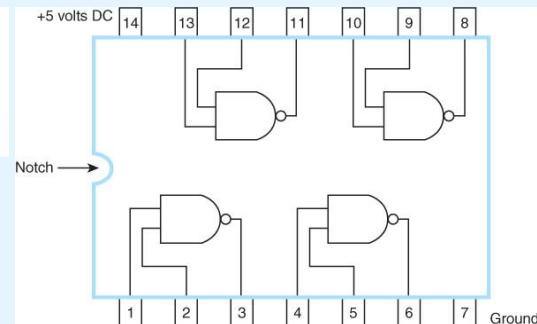
x	y	z	$x\bar{z} + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

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

### 3.3 Logic Gates

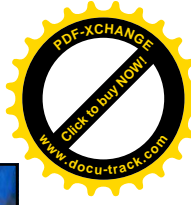
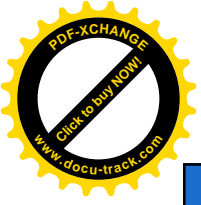
- We have looked at Boolean functions in abstract terms.
- In this section, we see that Boolean functions are implemented in digital computer circuits called gates.
- A gate is an electronic device that produces a result based on two or more input values.
  - In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
  - Integrated circuits contain collections of gates suited to a particular purpose.

### 3.3 Logic Gates



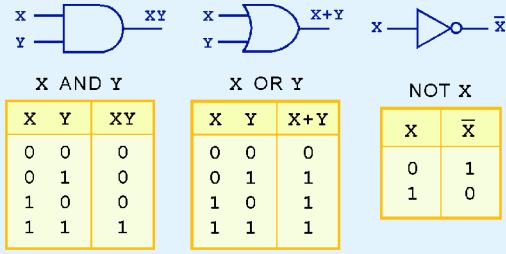
The small metal 'teeth' are the i/p & o/p pins. In the diagram to the right, they are the numbered boxes.

Simple SSI (small scale integration) Integrated Circuit Chip



### 3.3 Logic Gates

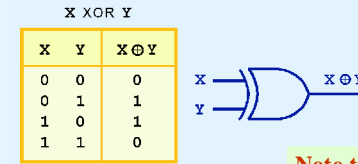
- The three simplest gates are the AND, OR, and NOT gates.



- They correspond directly to their respective Boolean operations, as you can see by their truth tables.

### 3.3 Logic Gates

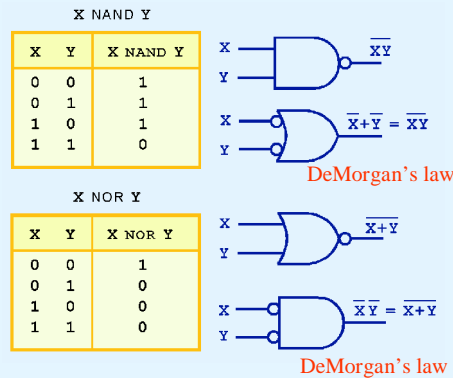
- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.



Note the special symbol  $\oplus$  for the XOR operation.

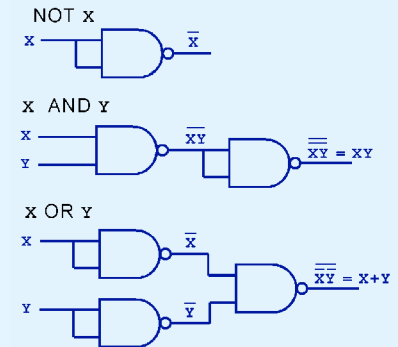
### 3.3 Logic Gates

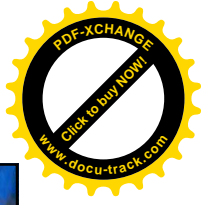
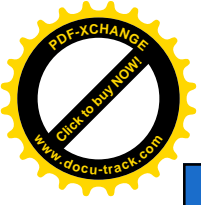
- Often, it is more economical to build gates that output the negated values rather than the original values directly.
- NAND and NOR are two very important gates. Their symbols and truth tables are shown at the right.



### 3.3 Logic Gates

- 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.
- The small circles on the Boolean symbols act as the NOT part of the Boolean operation.





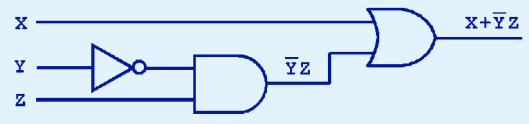
### 3.3 Logic Gates

- 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.



### 3.4 Digital Components

- The main thing to remember is that combinations of gates implement Boolean functions.
- The circuit below implements the Boolean function:  $F(X, Y, Z) = X + \bar{Y}Z$



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

### 3.5 Combinational Circuits

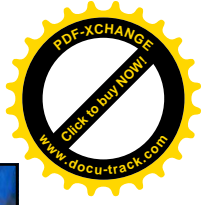
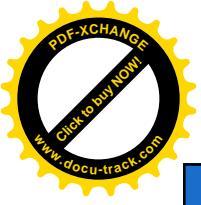
- We have designed a circuit that implements the Boolean function:  $F(X, Y, Z) = X + \bar{Y}Z$
- This circuit is an example of a *combinational logic* circuit.
- Combinational logic circuits produce a specified output (almost) at the instant when input values are applied. ***Its output is decided by the input only.***

In a later section, we will explore circuits where this is not the case.

### 3.5 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.
- The circuit logic is shown on the next slide.

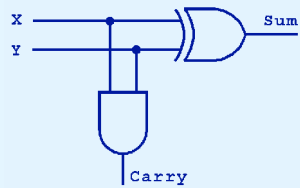
Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



### 3.5 Combinational Circuits

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

Sum =  $X \oplus Y$   
 Carry =  $XY$



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

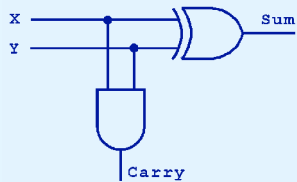
### 3.5 Combinational Circuits

- The half adder only adds two single-bit numbers.
- To add multiple bit numbers, we change our half adder into 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

### 3.5 Combinational Circuits

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

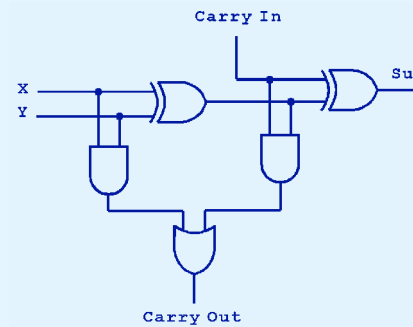


Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

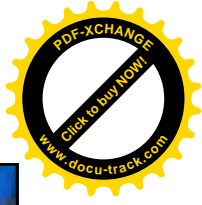
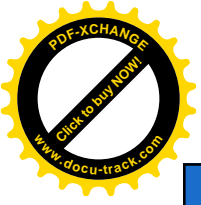
Note that on page 124 of the book, it does not include a 'carry in' column for a stand-alone half-adder, which makes more sense.

### 3.5 Combinational Circuits

- Here is a full adder – using two half adders and an OR gate.

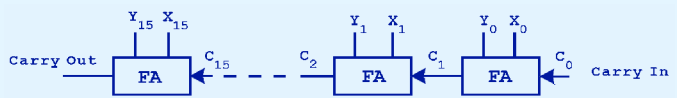


Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



### 3.5 Combinational Circuits

- Just as we combined half adders to make a full adder, full adders can be connected in series.
- The carry bit “ripples” from one adder to the next; hence, this configuration is called a *ripple-carry adder* (FA below means ‘Full Adder’). The diagram below is a ‘logic-diagram’.



Today’s systems employ more efficient adders.

### 3.5 Combinational Circuits

- **Decoders** are another important type of combinational circuit. It decodes binary information from a set of  $n$  inputs to a maximum of  $2^n$  outputs. A decoder uses the inputs and their respective values to select one specific output line. What do we mean by ‘select an output line’? It means that one unique output line is asserted, or set to a binary 1, while the other output lines are set to 0.
- Decoders are normally defined by the number of inputs and outputs – for example, a decoder with 3 inputs and 8 outputs is called a ‘3-to-8 decoder’ (3 binary bits = 8 unique bit combinations).

### 3.5 Combinational Circuits

- Among other things, they are useful in selecting a memory location based on a binary value placed on the address lines of a memory bus.
- Address decoders with  $n$  inputs can select any of  $2^n$  locations. For instance, if dealing with 1 MB of memory (as in a 16 bit environment), a 20 bit address is needed to be able to address any byte in this 1 MB. So a 20 bit input ( $n=20$ ) will give you any one of 1,048,576 (1 MB, which is  $2^{20}$ ) outputs.
- See pages 126-127 in the book for more details.

This is a block diagram for a decoder.



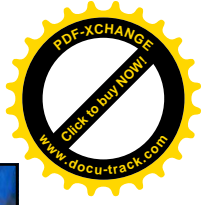
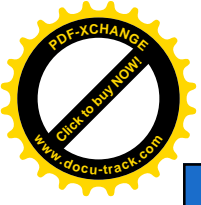
### 3.5 Combinational Circuits

- Let  $n=2$ , then  $2^n=4$ . The circuit becomes a 2-to-4 decoder (2 inputs; 4 outputs).
- $Z_0 = x'y = 1$
- $Z_1 = x'y = 1$
- $Z_2 = xy = 1$
- $Z_3 = xy = 1$

x	y	Z <sub>0</sub>	Z <sub>1</sub>	Z <sub>2</sub>	Z <sub>3</sub>
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

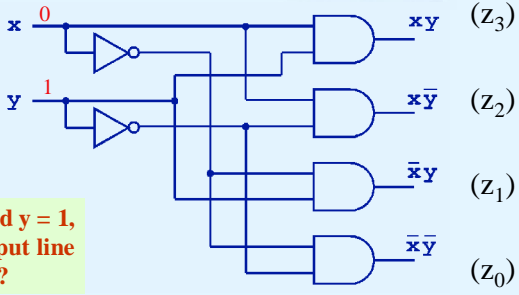
e.g. if you have 4 bytes of RAM, the 2 bit i/p x & y determines which byte # (address) of RAM you are addressing).

Truth table for a 2-to-4 decoder



### 3.5 Combinational Circuits

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

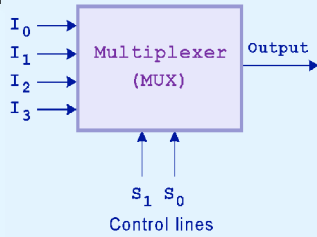


If  $x = 0$  and  $y = 1$ , which output line is enabled?

Answer:  $Z_1$  – This combination only ‘enables’ (turns to binary 1)  $Z_1$  (see previous slide).

### 3.5 Combinational Circuits

- 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, a.k.a. ‘selection variables’.
  - To be able to select among  $n$  inputs,  $\log_2 n$  control lines are needed (log function discussed on next slide).
- e.g., say you have many computers vying for 1 internet connection – the many computers are the ‘input’, and the single output determines which computer will be connected to this internet connection now. In this picture, we have 4 computers.



This is a block diagram for a multiplexer.

### 3.5 Combinational Circuits

$\log$  is a mathematical function. It is needed to determine how many control lines (bits) are needed to represent the number of i/p lines we have.

Here, we need 2 bits, to represent 00, 01, 10, & 11. If we had 16 i/p lines, we’d need 4 control lines. The formula is: take the number of i/p lines and keep dividing by 2 until (and including) you get a result of 1. The number of divisions done is your answer for number of control lines needed.

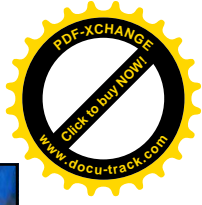
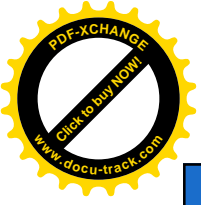
Divide 16/2, then 8/2, then 4/2, then 2/2. Quotient now =1. Four divisions were done, so 4 control lines are necessary.

### 3.5 Combinational Circuits

- Let  $n = 4$ , then  $\log_2 n = 2$  (i.e., 2 ‘control lines’). The circuit becomes a 4-to-1 multiplexer.
- Output =  $S'_1 S'_0 I_0 + S'_1 S_0 I_1 + S_1 S'_0 I_2 + S_1 S_0 I_3$
- The 4 possible control line values (2 bits means 4 possible values, 0 – 3), determine which one of the 4 i/p lines are chosen.
- $\log_2 n$  is = to the number of bits needed to represent ‘n’ number of bit combinations, e.g., if  $n=8$ ,  $\log_2 n = 3$ ; if  $n = 16$ ,  $\log_2 n = 4$  (to represent 16 different values).

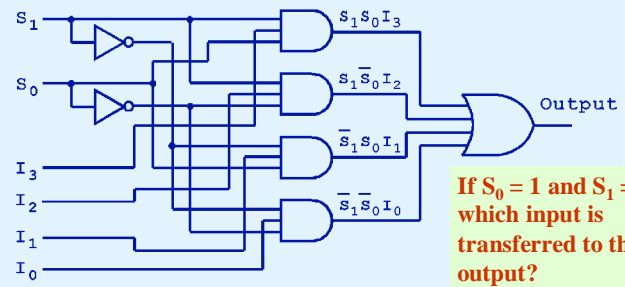
$S_1$	$S_0$	Output
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

Characteristic table for a 4-to-1 Multiplexer



### 3.5 Combinational Circuits

- 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?

Answer:  $I_1$  – both  $S_0$  &  $S_1$  must be = 1 at the i/p point to the AND symbol. This is true for the AND that the  $I_1$  is pointing to (2<sup>nd</sup> from the bottom).

### 3.5 Combinational Circuits

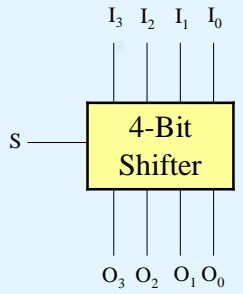
- An example of using a multiplexer is having four computers networked to one internet connection, where only one computer at a time can use it. The various computers are the ‘input’, and the single ‘output’ is the selected computer which will now be able to use this internet connection. The 2 bit control lines value determines which of the four computers will be ‘transferred’ to the output.

### 3.5 Combinational Circuits

- This 4-bit shifter moves the bits of a nibble one position to the left ( $S=0$ ) or right ( $S=1$ ).  $S$  is the ‘control line’. When you shift left, each bit moves one bit left, you lose the hi-order bit, and the low order bit gets a zero. When you shift right, you lose the low-order bit and the hi-order bit gets a zero.

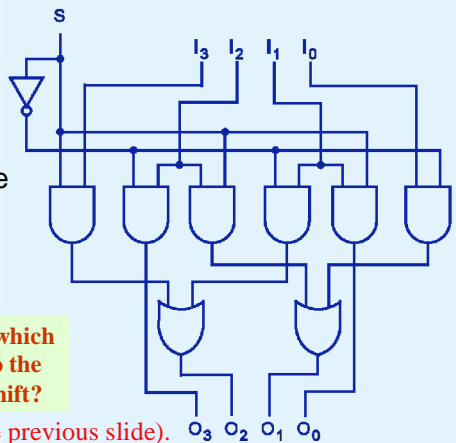
S	$O_3$	$O_2$	$O_1$	$O_0$
0	$I_2$	$I_1$	$I_0$	0
1	0	$I_3$	$I_2$	$I_1$

- $O_0 = SI_1$
- $O^1 = S'I_0 + SI_2$
- $O_2 = S'I_1 + SI_3$
- $O_3 = S'I_2$



### 3.5 Combinational Circuits

- This shifter moves the bits of a nibble one position to the left or right.



If  $S = 0$ , in which direction do the input bits shift?

Answer: Left (see previous slide).

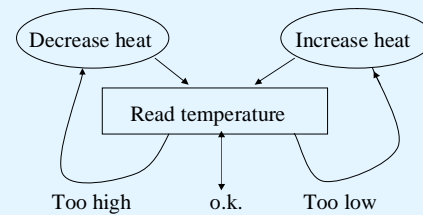
## 3.6 Sequential Circuits

- Combinational logic circuits are perfect for situations when we require the immediate application of a Boolean function to a set of inputs, **and all the inputs are coming in together.**
- 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.

49

## 3.6 Sequential Circuits

- Sequential circuits have ‘loops’ – these enable circuits to receive feedback. A simple example of a system dependent on feedback is a thermostat. It controls the temperature by receiving feedback in the form of a temperature reading. This system has loops. The output of the temperature changing function affects the input of the temperature-checking function. The output of the temperature-checking function affects the operation of the temperature changing function.



50

## 3.6 Sequential Circuits

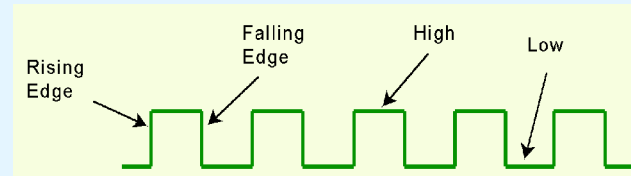
- As the name implies, sequential logic circuits require a means by which events can be sequenced.
- State changes are controlled by clocks.
  - A “clock” is a special circuit that sends electrical pulses through a circuit.
- Clocks produce electrical waveforms such as the one shown below.



51

## 3.6 Sequential Circuits

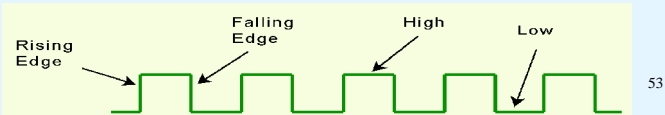
- State changes occur in **Synchronous** sequential circuits only when the clock ticks, i.e., a clock cycle. **There are Asynchronous sequential circuits, which are affected by any input value change. We will be discussing Synchronous sequential circuits only.**
- Circuits can change state on the rising edge, falling edge, or when the clock pulse reaches its highest voltage.



52

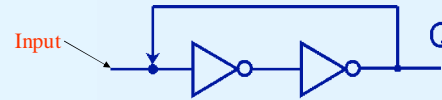
## 3.6 Sequential Circuits

- Circuits that change state on the rising edge, or falling edge of the clock pulse are called *edge-triggered*. **Flip-flops** (which we will get to soon), are edge-triggered. **Latches** are level triggered (whether high or low). Many people use the term interchangeably. We will use the term *flip-flops* exclusively.
- *Level-triggered circuits* change state when the clock voltage reaches its highest or lowest level.
- As far as our we are concerned, this is just informational; it does not affect our discussion of circuit design.



## 3.6 Sequential Circuits

- 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. The feedback reinforces the input, *so that even if the input is removed, the output will remain*. This is not a useful circuit, but shows how feedback works.



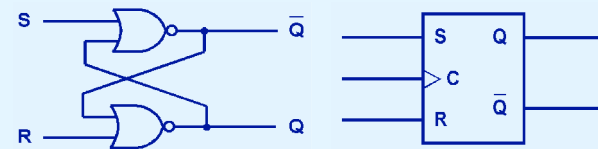
## 3.6 Sequential Circuits

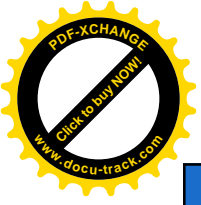
- Let's use a soda machine as an example. You deposit a nickel. When you deposit the next coin, the machine must 'remember' that a nickel has already been deposited, so assuming that you have a bit to represent that nickel, it must be *set* to a binary 1 before the next coin is deposited – i.e., it needs to 'remember' this via 'feedback'. If you press the refund button, the bit should *reset* to a binary 0, so that the when the next coin is deposited, the machine shouldn't think that a nickel has already been deposited.
- You will need a number of flip flops in order to be able to represent all the possible 'states' the soda machine is in, i.e., how much money has been deposited at any given time.

55

## 3.6 Sequential Circuits

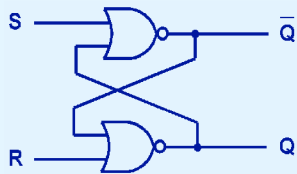
- You can see how feedback works by examining the most basic sequential logic components, the SR flip-flop. It has two inputs: S and R.
  - The "SR" stands for set/reset.
- It has two outputs – Q and Q'. A clock is supplied to start the change of the state of the flip-flop. 'Q' is always the 'current' state.
- The internals of an SR flip-flop are shown below, along with its block diagram.



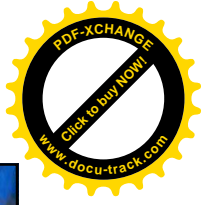


### 3.6 Sequential Circuits

- The behavior of an SR flip-flop is described by a characteristic table.
- $Q(t)$  means the value of the output at time 't', i.e., the **current state**.  $Q(t+1)$  is the value of Q after the next clock pulse.
- $Q$  &  $Q'$  must always be opposites, i.e., one = 0 and the other = 1.  $Q'$  does not mean zero; it means the complement of Q.
- If after the first go-round, if you turn off the S & R inputs, the flip-flop will still keep 're-enforcing its value for Q & Q'.



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



### 3.6 Sequential Circuits

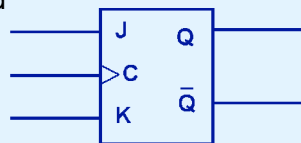
- The SR flip-flop actually has three inputs: S, R, and its current output, Q.
- Thus, we can construct a truth table for this circuit, as shown at the right.
- Notice the two undefined values. When both S and R are 1, the SR flip-flop is unstable, i.e., the output may have both  $Q$  &  $Q' = 0$ , which makes this circuit 'unpredictable' – as Q and Q' must be opposites.

Present State			Next State
S	R	Q(t)	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

58

### 3.6 Sequential Circuits

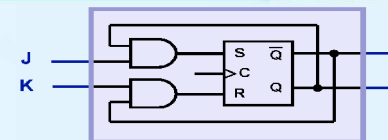
- If we can be sure that the inputs to an SR flip-flop will never both be 1, we will never have an unstable circuit. This may not always be the case.
- The SR flip-flop can be modified to provide a stable state when both inputs are 1 (this means that the inputs to J & K are both 1; the addition of the two AND gates guarantees that S & R will never both be = 1).
- This modified flip-flop is called a JK flip-flop, shown at the right – explained more fully on next slide.
- The "JK" is in honor of Jack Kilby, a Texas Instruments engineer who invented the integrated circuit in 1958.



59

### 3.6 Sequential Circuits

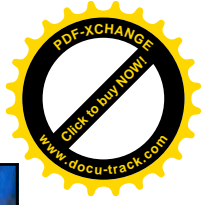
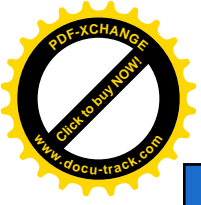
- At the right, we see how an SR flip-flop can be modified to create a JK flip-flop. Note that there is an SR flip-flop inside.
- The characteristic table indicates that the flip-flop is stable (i.e., predictable value) for all inputs. Note that the first three values are the same as regular SR flip-flops.
- Since Q & Q' are each fed into an AND gate, it is not possible to have two binary '1's feeding into the SR flip-flop inside.



In order to 'walk-thru' this gate, you'd have to substitute the block diagram for the SR above with the SR gate diagram from slide 57, above.

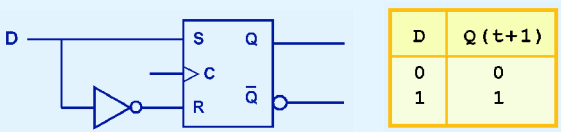
J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	Q(t) i.e., toggle

60



### 3.6 Sequential Circuits

- 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.

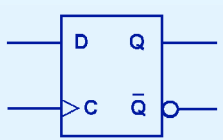


D	Q(t+1)
0	0
1	1

61

### 3.6 Sequential Circuits

- The D flip-flop is the fundamental circuit of computer memory (RAM).
  - D flip-flops are usually illustrated using the block diagram shown below.
- The characteristic table for the D flip-flop is shown at the right.



D	Q(t+1)
0	0
1	1

62

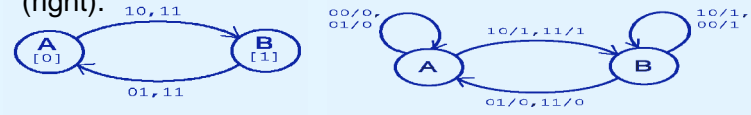
### 3.6 Sequential Circuits

- The behavior of sequential circuits can be expressed using characteristic tables. **Another method is to express this behavior is via finite state machines (FSMs) (this is not hardware; it's just a representation of the behavior of sequential circuits)**.
  - FSMs consist of a set of nodes that hold the states of the machine and a set of arcs that connect the states.
- Moore and Mealy machines are two types of FSMs that are equivalent.
  - They differ only in how they express the outputs of the machine.
- Moore machines place outputs on each node, while Mealy machines present their outputs on the transitions.

63

### 3.6 Sequential Circuits

- The behavior of a JK flop-flop is depicted below by a **simplified** Moore machine (left) and a Mealy machine (right).



A & B 'nodes' show the possible **current state** values (what we called 'Q' earlier). The Moore machine doesn't show the i/p values for the 'no-change' state. The 10,11 from A to B are the J & K values which would 'set' the value from 0 to 1; the 01,11 are the J & K values to 'reset' the value from 0 to 1 (see the characteristic table on slide 60).

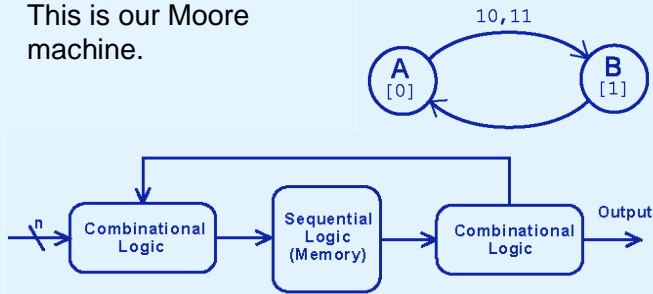
The Mealy machine shows the same concept in a slightly different format. The circles above the nodes are 'reflexive arcs'. For instance, the above shows the i/p values which would keep 'A' a zero – 00 or 01, and the result following the slash. Same idea for 'B'. The arcs work the same as the Moore machine, except that the binary result is also shown following the slash.

64

### 3.6 Sequential Circuits

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

This is our Moore machine.

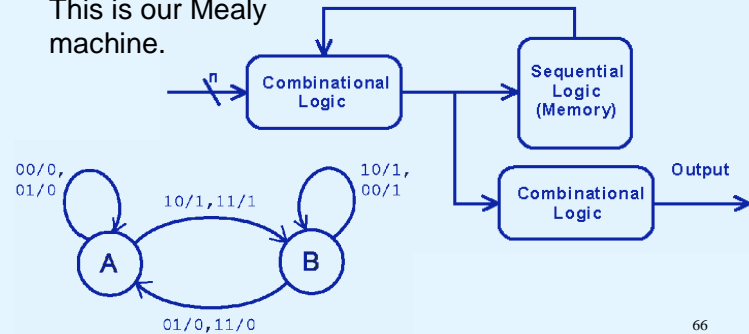


65

### 3.6 Sequential Circuits

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

This is our Mealy machine.



66

### 3.6 Sequential Circuits

- The graphical models used for Moore & Mealy machines are useful for high-level conceptual modeling of the behavior of circuits. However, once a circuit reaches a certain level of complexity, they become unwieldy and only with great difficulty capture the details required for implementation.

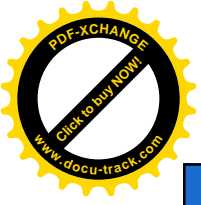
67

### 3.6 Sequential Circuits

- It is difficult to express the complexities of actual implementations using only Moore and Mealy machines.
  - For one thing, they do not address the intricacies of timing very well.
  - Secondly, it is often the case that an interaction of numerous signals is required to advance a machine from one state to the next.
- For these reasons, Christopher Clare invented the algorithmic state machine (ASM).

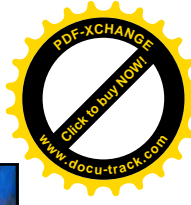
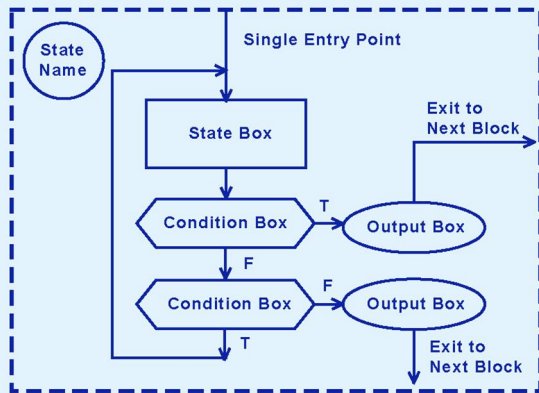
The next slide illustrates the components of an ASM.

68



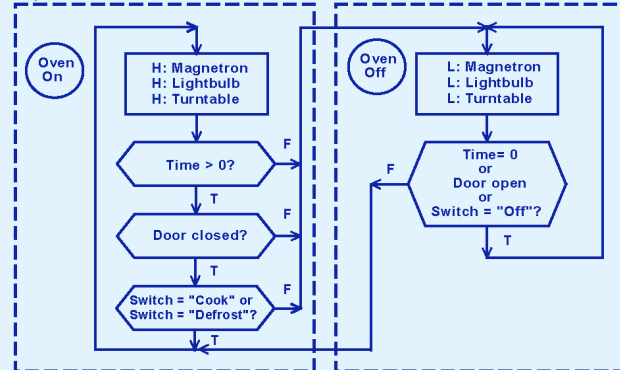
### 3.6 Sequential Circuits

State Block



### 3.6 Sequential Circuits

- This is an ASM for a microwave oven. 'H' means the line is 'on'; 'L' means the line is 'off'.



### 3.6 Sequential Circuits

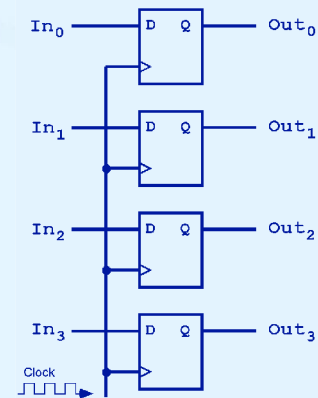
- Sequential circuits are used anytime that we have a "stateful" application.
  - A stateful application is one where the next state of the machine depends on the current state of the machine and the input.
- A stateful application requires both combinational (gates) and sequential (flip-flops) logic.
- The following slides provide several examples of circuits that fall into this category.

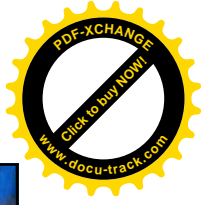
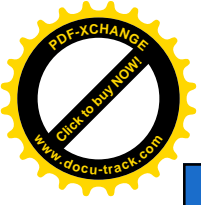
### 3.6 Sequential Circuits

- 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 shown on the next slide.

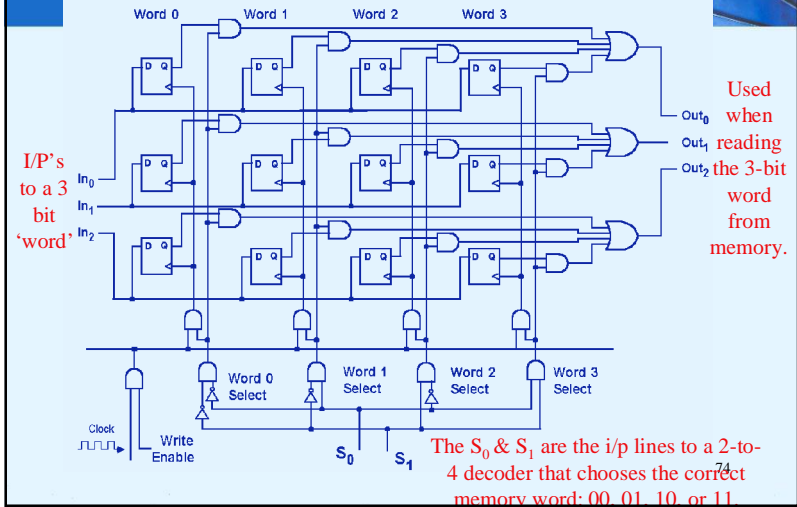




## 3.6 Sequential Circuits

- A detailed explanation of the logic diagram on the next slide is given in the book on pages 141-142.
- It represents the logic to write a three bit 'word' to memory.

## 3.6 Sequential Circuits



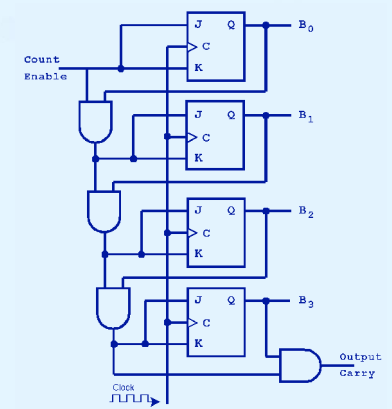
## 3.6 Sequential Circuits

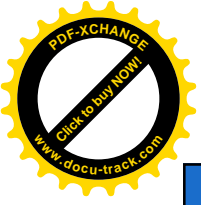
- A binary counter is another example of a sequential circuit.
- The low-order bit ( $B_0$ ) is complemented at each clock pulse.
- Whenever it changes from 1 to 0, the next bit is complemented, and so on through the other flip-flops.

	Col. 1	Col. 2
$B_3B_2B_1B_0$	$B_3B_2B_1B_0$	$B_3B_2B_1B_0$
	0000	1000
	0001	1001
	0010	1010
	0011	1011
	0100	1100
	0101	1101
	0110	1110
	0111	1111

## 3.6 Sequential Circuits

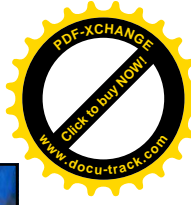
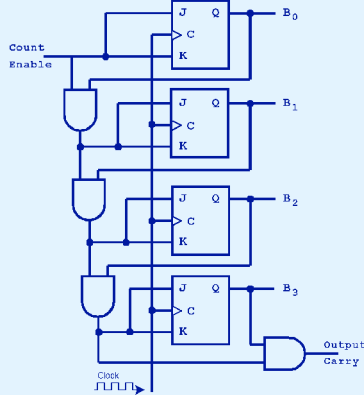
- The 'output' values ( $B_0, B_1$ , etc.) get their values in 1 clock cycle, but are not fed into the next bit until the next clock cycle. So, e.g., the value for  $B_1$  for any given clock cycle will be determined by its input values from the: 1) clock, 2) count enable line, 3) the value from  $B_0$  (which was set on the previous clock cycle).





### 3.6 Sequential Circuits

- Because of the complementing states, the counter is best implemented using JK flip-flops. Recall that the JK flip-flop changes state when J and K are both = 1.

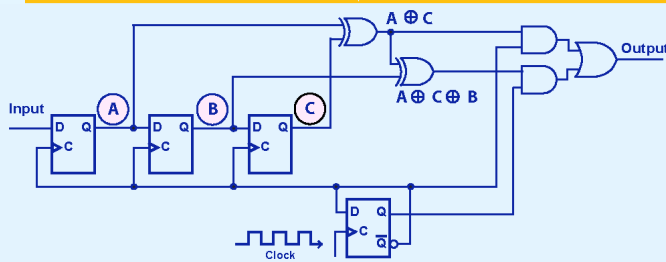


### 3.6 Sequential Circuits

- Convolutional coding and decoding requires sequential circuits.
- One important convolutional code is the (2,1) PRML (partial response maximum likelihood) code that is briefly described at the end of Chapter 2.
- A (2, 1) convolutional code is so named because two symbols are output for every one symbol input (the size of the encoded data stream is twice the size of the original one).
- A convolutional encoder for PRML with its characteristic table is shown on the next slide.

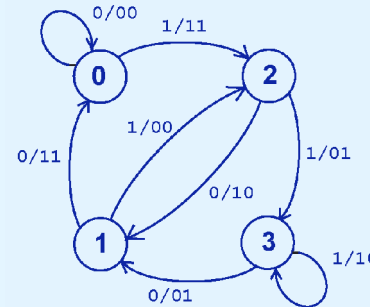
### 3.6 Sequential Circuits

Input A	Current State B C	Next State B C	Output	Input A	Current State B C	Next State B C	Output
0	00	00	00	0	10	01	10
1	00	10	11	1	10	11	01
0	01	00	11	0	11	01	01
1	01	10	00	1	11	11	10

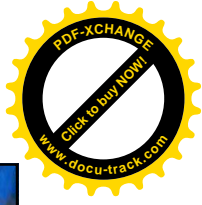
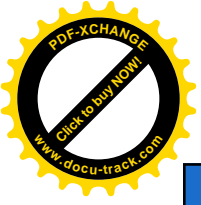


### 3.6 Sequential Circuits

This is the Mealy machine for our encoder.

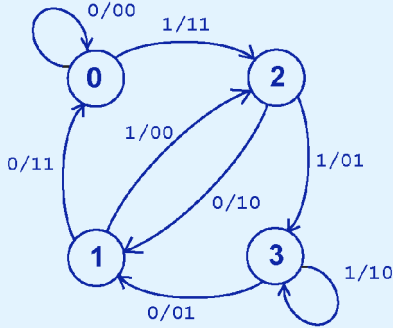


Input A	Current State B C	Next State B C	Output
0	00	00	00
1	00	10	11
0	01	00	11
1	01	10	00
0	10	01	10
1	10	11	01
0	11	01	01
1	11	11	10



### 3.6 Sequential Circuits

- The fact that there is a limited set of possible state transitions in the encoding process is crucial to the error correcting capabilities of PRML.
- You can see by our Mealy machine for encoding that:

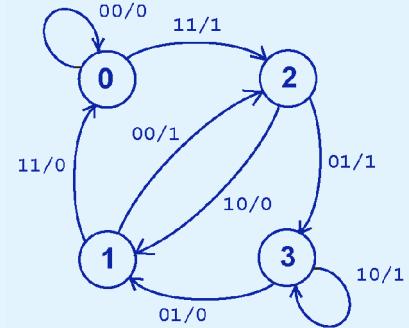


$$F(1101\ 0010) = 11\ 01\ 01\ 00\ 10\ 11\ 11\ 10.$$

81

### 3.6 Sequential Circuits

- The decoding of our code is provided by inverting the inputs and outputs of the Mealy machine for the encoding process.
- You can see by our Mealy machine for decoding that:

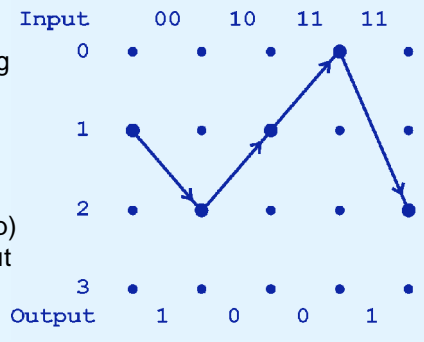


$$F(11\ 01\ 01\ 00\ 10\ 11\ 11\ 10) = 1101\ 0010$$

82

### 3.6 Sequential Circuits

- Yet another way of looking at the decoding process is through a lattice diagram.
- Here we have plotted the state transitions based on the input (top) and showing the output at the bottom for the string 00 10 11 11.

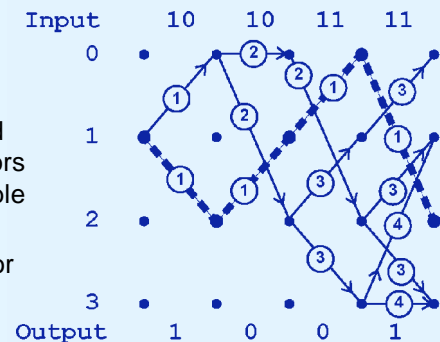


$$F(00\ 10\ 11\ 11) = 1001$$

83

### 3.6 Sequential Circuits

- Suppose we receive the erroneous string: 10 10 11 11.
- Here we have plotted the accumulated errors based on the allowable transitions.
- The path of least error outputs 1001, thus 1001 is the string of maximum likelihood.



$$F(00\ 10\ 11\ 11) = 1001$$

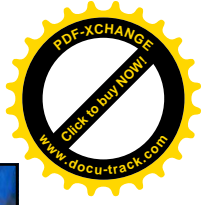
84



## 3.7 Designing Circuits

- We have seen digital circuits from two points of view: digital analysis and digital synthesis.
  - *Digital analysis* explores the relationship between a circuit's 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.

85



## 3.7 Designing Circuits

- 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.

86

## 3.7 Designing Circuits

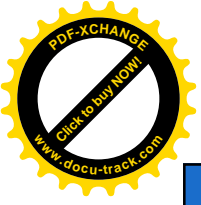
- When we need to implement a simple, specialized algorithm and its execution speed must be as fast as possible, a hardware solution is often preferred.
- This is the idea behind *embedded systems*, which are small special-purpose computers that we find in many everyday things.
- Embedded systems require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

87

## Chapter 3 Conclusion

- Computers are implementations of Boolean logic.
- Boolean functions are completely described by truth tables.
- Logic gates are small circuits that implement Boolean operators.
- The basic gates are AND, OR, and NOT.
  - The XOR gate is very useful in parity checkers and adders.
- The “universal gates” are NOR, and NAND.

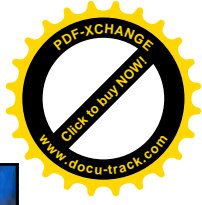
88



## Chapter 3 Conclusion

- Computer circuits consist of combinational logic circuits and sequential logic circuits.
- Combinational circuits produce outputs (almost) immediately when their inputs change.
- Sequential circuits require clocks to control their changes of state.
- The basic sequential circuit unit is the flip-flop: The behaviors of the SR, JK, and D flip-flops are the most important to know.

89



## Chapter 3 Conclusion

- The behavior of sequential circuits can be expressed using characteristic tables or through various finite state machines.
- Moore and Mealy machines are two finite state machines that model high-level circuit behavior.
- Algorithmic state machines are better than Moore and Mealy machines at expressing timing and complex signal interactions.
- Examples of sequential circuits include memory, counters, and Viterbi encoders and decoders.

90



91